NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A PROTOTYPE RAY TRACER

by

Paul Gerard Smith

June 1987

Thesis Advisor: Michael J. Zyda

unclassified
ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a REPORT SECURITY CLASSIFICATION: unclassified	1b RESTRICTIVE MARKINGS
a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
b DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
a NAME OF PERFORMING ORGANIZATION: aval Postgraduate School; 6b OFFICE SYMBOL (If applicable): 52	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
: ADDRESS (City, State, and ZIP Code): onterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
a NAME OF FUNDING/SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

TITLE (Include Security Classification)

A PROTOTYPE RAY TRACER

PERSONAL AUTHOR(S) Smith, Paul Gerard

a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
aster's Thesis	FROM _____ TO _____	1987 June	138

SUPPLEMENTARY NOTATION

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	illumination models, rendering, lighting and shading, and ray tracing

ABSTRACT (Continue on reverse if necessary and identify by block number)

he ability to make computer images more realistic is becoming more impor-ant as the hardware for producing such images is becoming less expensive nd hence more available. The key to producing realistic images lies in he algorithms that can take full advantage of the hardware to produce hem. In this study, we look at a prototype of a ray tracer, as presented n [Ref. 1]. Ray tracing, in combination with a global illumination odel, currently provides the most realistic images that can be generated n general purpose computing hardware. The prototype was successfully mplemented on an IBM AT clone.

D STRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: unclassified
a NAME OF RESPONSIBLE INDIVIDUAL: Prof. Michael J. Zyda	22b TELEPHONE (Include Area Code): (408) 646-2305; 22c OFFICE SYMBOL: Code 52Zk

D FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
unclassified

Approved for public release; distribution is unlimited.

A Prototype Ray Tracer

by

Paul Gerard Smith
Captain, United States Marine Corps
B. A., The Citadel, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1987

ABSTRACT

The ability to make computer images more realistic is becoming more important as the hardware for producing such images is becoming less expensive and hence more available. The key to producing realistic images lies in the algorithms that can take full advantage of the hardware to produce them. In this study, we look at a prototype of a ray tracer, as presented in [Ref. 1]. Ray tracing, in combination with a global illumination model, currently provides the most realistic images that can be generated on general purpose computing hardware. The prototype was succesfully implemented on an IBM AT clone.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to a number of people who helped him throughout this study. To my advisor, **Dr. Michael Zyda**, who provided me with the knowledge, insight, and encouragement necessary to complete the project. I also wish to express my appreciation for his patience and assistance in the writing of this study.

The following people provided the insight and knowledge which were incorporated into the project:

- **LCDR John Falby**, USN, for the data structure used in this study;
- **Dr. Maurice D. Wier** for help in understanding the fundamentals of vector calculus.

I also want to express special thanks to LT Dale Streyle, USCG, and CAPT Douglas Smith, USMC, for their help and support throughout my education at the Naval Postgraduate School. I appreciate the help they rendered from the very first project to running off the final copy of this thesis.

Last but not least I want to express my gratitude to my wife, Terri, and to my son, Joshua for their patience and support throughout my education at the Naval Postgraduate School. I thank them for enduring the long hours of study and my wife for editing my writing.

I. INTRODUCTION

From the beginning of recorded history mankind has always had the need to create pictures. The reasons for creating these pictures ranges from the aesthetic, pretty pictures are nice to look at, to the functional, pictures can be an excellent way to communicate information. As mankind progressed, so did his ability to create pictures, although the techniques used to create pictures basically stayed the same. The advent of computers gave man yet another tool with which to create pictures.

The rapid increase in technology has made computer graphics a rapidly growing field. For the first time since man started drawing pictures, completely new techniques needed to be developed. In computer graphics the brush, paint and canvas are replaced by the mouse, algorithm and display. However even though the tools have changed, the same problems remain: how to make the picture look better, be it either more pleasing to the eye or to get the information across more clearly.

Two of the most common and difficult problems in computer graphics are the hidden surface and lighting and shading problems. A large number of solutions exist to both of these problems. Very few solutions can be applied to both. One such solution is ray tracing. Ray tracing is the process of following an imaginary

ray from a viewpoint through a pixel on a screen and into a scene to determine if it intersects any objects in the scene and then calculating the intensity of the pixel it went through based on the the final destination of the ray. As in most cases where one solution is found to several problems, that solution is seldom the best for all the problems it is applied to and so it becomes a matter of trade-offs. Such is the case with ray tracing. Among the hidden surface removal techniques, ray tracing is the least efficient being referred to as a brute force technique. In contrast, it has been labeled as one of the most elegant techniques in regards to lighting and shading [Ref. 2: p. 137]. Because of this latter fact, ray tracing has become an important technique in computer graphics. Ever since the idea behind ray tracing was suggested by Appel, numerous articles, studies, and implementations have been done on it. These in turn have spawned fruther extensions and modifications. [Ref. 1: p. 296]

A. DEFINITION AND OVERVIEW

The idea behind ray tracing lies in the theory that the light in our environment can be modeled as rays. After being emitted from a source, the rays are then reflected and refracted through a scene. Some of the rays eventually find their way to the eye where the scene is recreated (Figure 1.1a). These light rays are emitted from light sources, such as the sun. An infinite number of light rays exist, but only a small percentage of them are received by us. To try and trace these rays from the source is computationally expensive. Appel suggested that

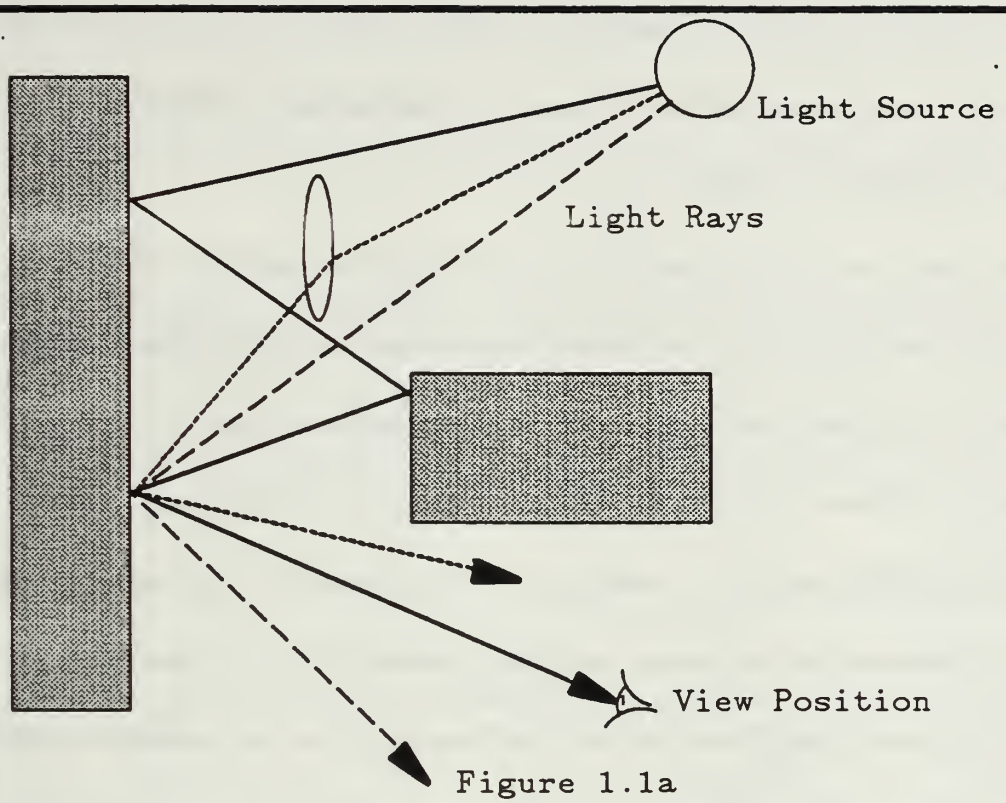

Figure 1.1a

Reaction of Light Rays with Objects [Ref. 3: p.53]

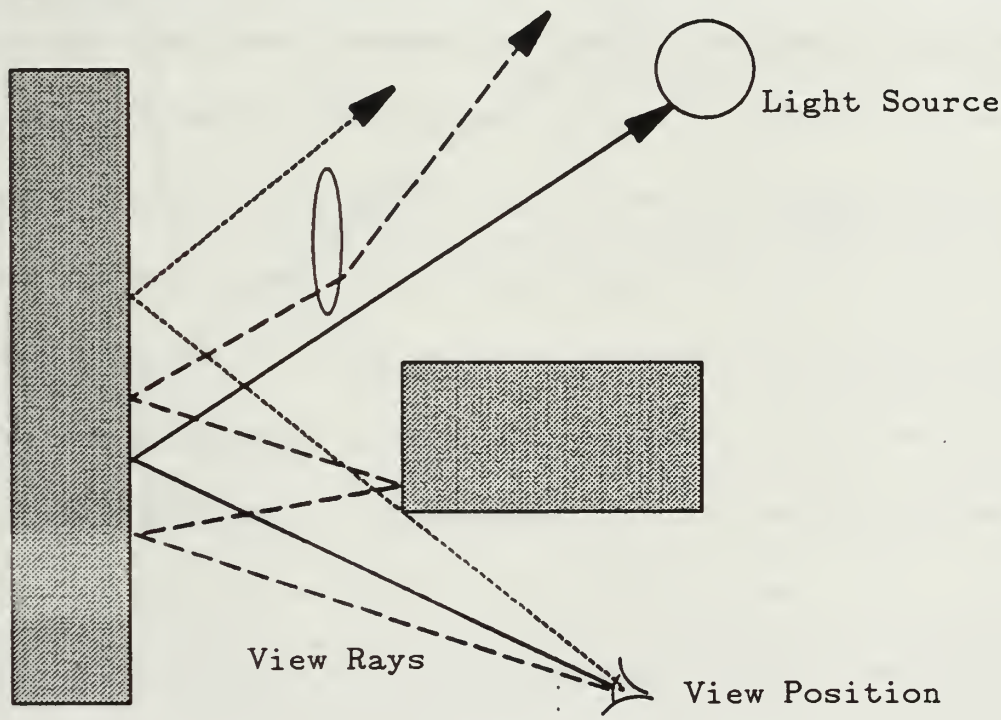

Figure 1.1b

Tracing Rays Backwards from View Position

instead of tracing the rays from the source that they should be traced backwards from the viewer, thus dealing with only those rays that actually contribute to the scene (Figure 1.1b). [Ref. 1: p. 296]

The basic ray tracing algorithm is a very simple one and not difficult to implement. The basic algorithm is a hidden surface algorithm. All hidden surface algorithms can be classified based on the coordinate system or space in which they are implemented. These are either in object space or in image space. The ray tracing algorithm falls under the category of image space. This category of algorithm is implemented in the screen coordinate system in which the objects are viewed. Unfortunately, the calculations are performed only to the precision of the scene representation, which generally provides poor resolution. The image space algorithms work by comparing every object in the scene with every pixel. Such an algorithm is computationally expensive. Ray tracing algorithms have three parts: a viewpoint, a raster screen, and a set of objects (Figure 1.2). In the algorithm, the viewpoint is along the positive z axis. From this point, a ray is shot into the scene through the center of every pixel on the raster. Each of these rays is then traced and compared against every object in the scene to determine if there is an intersection with any of them. It is in the determination of a possible intersection point that a ray tracer spends anywhere from 75 to 95 percent of its time. If there is an intersection, then the intensity at the pixel is determined using the intersected object's attributes and an appropriate illumination model. If there is no intersection, then the pixel intensity is determined by the background

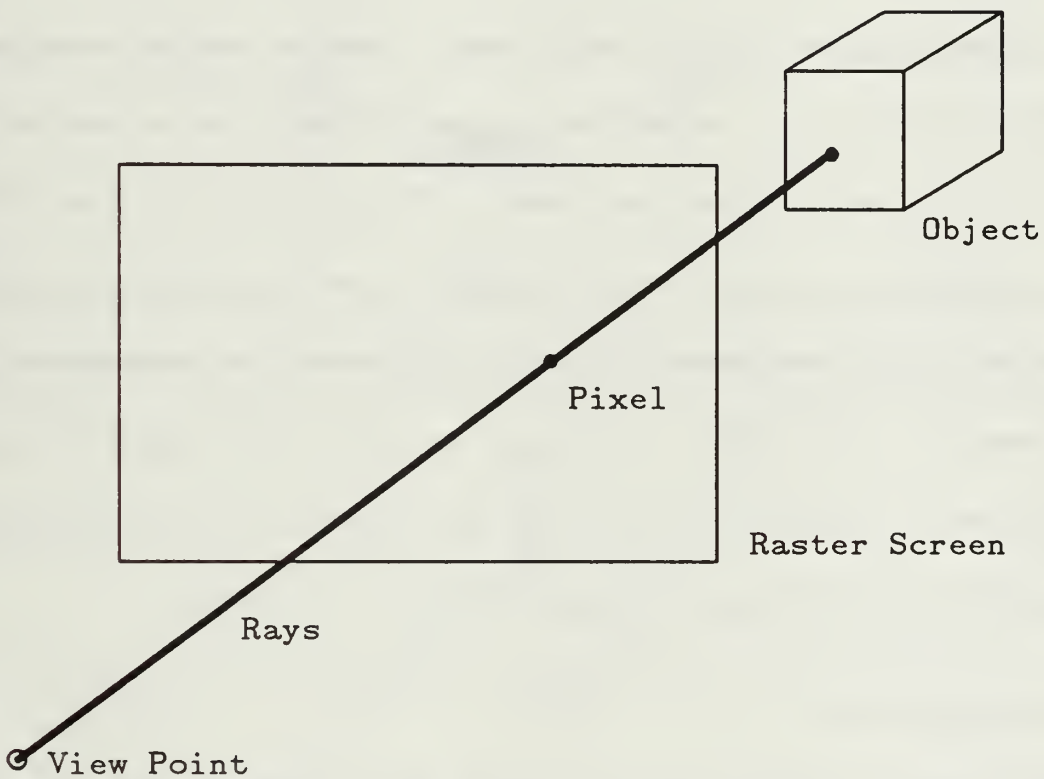

Figure 1.2
Layout of Ray Tracing Scene [Ref. 1: p.297]

intensity. This procedure is then repeated for every pixel on the raster. When the ray tracer is used as a hidden surface algorithm, intersection testing stops after the first intersection. Extensions to the basic ray tracing algorithm, that showed its usefulness in implementing a global illumination model, were originally implemented by Whitted [Ref. 4] and Kay [Ref. 5 and 6]. In these extensions of ray tracing, additional rays are calculated, specifically the reflected and refracted rays, and then tested to see if they intersect with any objects in the scene. This process of generating new rays and tracing them to check for possible intersections is continued until the rays either leave the scene or stack space is exceeded. In such a case, the remaining rays are treated as if they had left the scene. This process, illustrated in Figure 1.3a, for a single ray with intersections is easily represented using the tree structure shown in Figure 1.3b. Here each node of the tree represents a ray surface intersection. At each node, at least one and sometimes two subbranches are generated. One branch of each of the reflected and refracted rays is generated from the point. [Ref. 1: pp. 190-296]

B. ORGANIZATION

This study is broken into three areas: data requirements, ray tracing methodology, and the intensity problem. The first section reviews the data needed for a lighting and shading modeler, hereafter referred to as a renderer, of which a ray tracer is an integral part. The second section reviews the actual process of tracing a ray through a scene to be rendered. The third section looks briefly at the

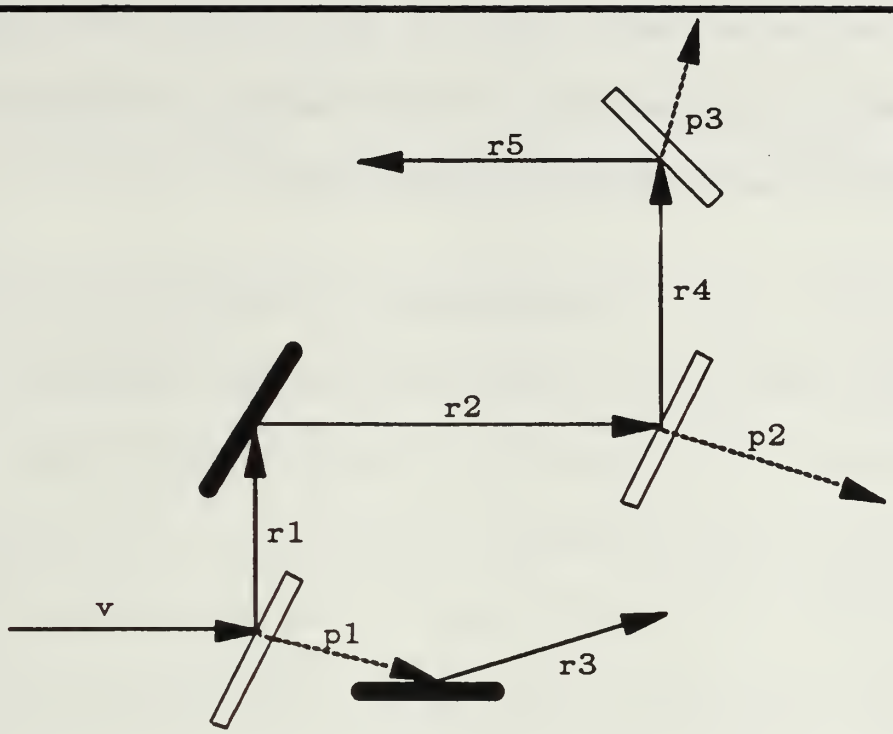

Figure 1.3a

Obtaining Global Data Reflection & Transmission Rays [Ref. 3: p.61]

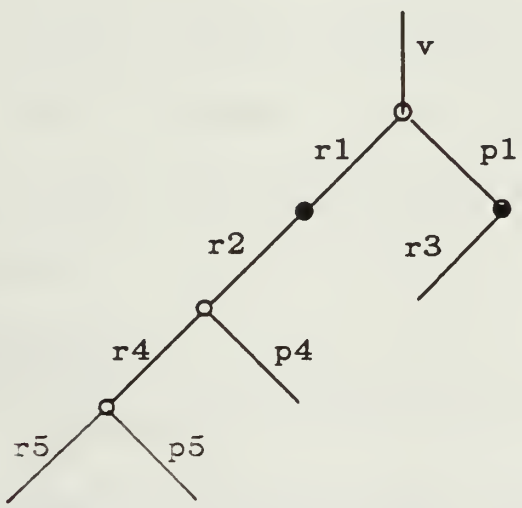

Figure 1.3b

Global Data Tree [Ref. 3: p.62]

illumination problem and how it relates to the ray tracer. The concluding chapters present the implementation, and known limitations of the model along with areas of future research.

II. DATA REQUIREMENTS

A. OVERVIEW OF THE DATA REQUIREMENTS

The importance of the intersection routines in the ray tracer is apparent in the fact that a ray tracer spends 75 to 95 percent of its time determining intersections [Ref. 1: p. 297]. The key to determining intersections, however, lies in large part on the data used to describe the scene that is being rendered. Information is needed not only to describe the entire scene that is being rendered but more importantly to describe each object in the scene. Scene data is that information needed to completely describe a picture, i.e., the number, kind, shape, and color of any objects in the picture along with the background intensity and light source information. This information must be properly ordered and broken down. Falby [Ref. 3] suggested that a scene be broken into three categories: object, view, and light. Each of these areas is examined below in the context of a ray tracing algorithm.

1. Object Data

The data pertaining to each object in the scene can be grouped into two categories: polygon and bounding volume. The reason for this breakdown is twofold. First, each object in the scene is composed of polygons. They are the basic building blocks of the scene. Second, in order to reduce the number of

intersection checks, it is necessary to set up a boundary around each object so that the ray tracer only performs the intersection checks in the regions that actually contain an object. Such a boundary is called a bounding volume. We examine the polygon data first.

a. Polygon Data

Since the main focus of the ray tracing algorithm lies in determining the intersections between the rays shot into a scene and the objects that make up a scene, and since each object is comprised of polygons, the problem is really one of determining the intersection points between the rays and the polygons. From the fundamentals of vector calculus, it is known that in order to determine the intersection between a ray and a polygon only the vertices of the polygon are needed as well as the direction of the ray and a point on the ray. Since the object is to be constructed of polygons, its vertices are known. Therefore, it is only necessary to ensure that these points are stored in some manner, such as a record, so as to be accessible to the ray tracer. In order to determine what the intensity of the pixel is through which the ray passes, it is essential that the characteristics of the object whose polygon was intersected be available. Since an object is made of polygons, they inherit the characteristics of the object. These characteristics also need to be readily accessible and, therefore, need to be stored in some manner. The following is a list of the basic object characteristics that need to be available: (1) the specular, diffuse, and transmission coefficients; (2) the Phong specular exponent; and (3) the index of refraction, see Table 2.1. [Ref. 3: p. 68]

TABLE 2.1: OBJECT DATA

FIELD NAME	VARIABLE NAME	VALUE
Polygon Vertices	x, y, z	real
Diffuse Coefficient	$Kd_{r,g,b}$	real (0-1)
Specular Coefficient	$Ks_{r,g,b}$	real (0-1)
Transmission Coefficient	$Kt_{r,g,b}$	real (0-1)
Unit Surface Normal	x, y, z	real (0-1)
Phong Specular Exponent	n	integer (0-200)
Index of Refraction	η_2	real

b. Bounding Volume Data

The major disadvantage of ray tracing is that it takes so much time. This is hard to avoid since it is so computationally expensive. It is essential, therefore, that more efficient techniques be developed to assist in reducing the number of calculations. Several techniques already exist with the bounding volume being the most effective [Ref. 1: p. 298]. In the description of ray tracing given so far, it has been stated that a ray is checked to see if it intersects with any object. Upon dissecting this statement further, a better understanding of the intersection problem can be realized. Unless some optimization is done, the ray tracing algorithm is forced to do the following. Each ray must be checked for a possible intersection with each object. Since each object is made up of polygons, then the ray must be checked for a possible intersection with each polygon. For a complicated object, such as a teapot, this requires a large number of checks and must be done for each object. The purpose for establishing the bounding volume lies in two facts. The first is that generally scenes are mostly background with just a few objects, hence very few of the rays actually hit anything. Therefore,

most of the intersection tests done are a waste of time. Second, a ray can only hit one object at a time. To have it process through the entire list of objects, when intersections with most of them can be eliminated, is needless. A bounding volume is, therefore, a method of enclosing each object in the scene in a simple containment vessel, which in effect creates a boundary around the object. Once this boundary is established, the number of overall intersection tests can be greatly reduced, as in the example of a teapot, which might easily have over a hundred polygons. If it is surrounded by a bounding box consisting of just six polygons, the number of intersection tests can be significantly reduced. In this situation, instead of having to test each ray against each polygon of the object, only those rays that penetrate the bounding volume need to be checked. Thus the bounding volume is a way to filter out unnecessary intersection tests by limiting the tests to those rays that are most likely to intersect an object.

Just as the use of a bounding volume greatly increases the efficiency of the ray tracing algorithm, the use of the right kind of a bounding volume can improve upon that even more. In Rogers [Ref. 1], the bounding volumes suggested are a bounding box and a bounding sphere (Figure 2.1), each of which has advantages and disadvantages. The bounding sphere is much easier to implement although it is less efficient in reducing the target area than is the bounding box, see Figure 2.1. The bounding box, on the other hand, is computationally expensive to implement. The data needed to establish a bounding sphere is minimal. It only requires a center point for the object and a

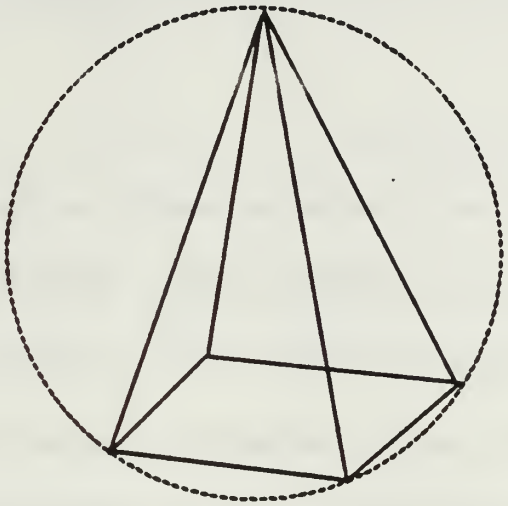

Figure 2.1a
Object Surrounded by Bounding Sphere [Ref. 1: p.298]

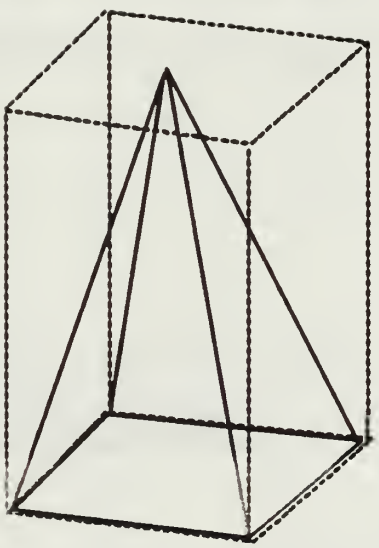

Figure 2.1b
Object Surrounded by Bounding Box [Ref. 1: p.298]

radius that encompasses every point of the object. The bounding box, on the other hand, requires far more data in that the polygons that make up the box must be described.

2. View Data

In rendering any scene, certain information can be applied to the scene as a whole. This information is grouped together to form the view data. This data consists of the viewpoint position, a constant to prevent division by zero, a refraction index for the global medium, the ambient light intensity, the background color, and the scene dimensions, see Table 2.2. [Ref. 3: pp. 74-75]

3. Light Data

To support a lighting and shading model, it is necessary to include certain information on the light source for the scene. That information must include the position of the light source, its intensity, its type, geometry, and dimension, see Table 2.3.

TABLE 2.2: VIEW DATA

FIELD NAME	VARIABLE NAME	VALUES
Viewpoint	x, y, z	real
No Zero Constant	Ko	real (<0)
Global Refraction Index	η_1	real
Ambient Light	$Ia_{r,g,b}$	real (0-1)
Background Light	$Ib_{r,g,b}$	real (0-1)
Scene Size	x, y	integers

TABLE 2.3: LIGHT DATA

FIELD NAME	VARIABLE NAME	VALUE
Light Position	x, y, z	real
Intensity	$I_{r,g,b}$	real (0-1)
Type	type	enumerated (point,distributed)
Shape	shape	enumerated (circular, rectangular)
Dimensions	x, y	real

B. DATA STRUCTURE FOR A RAY TRACER

Falby [Ref. 3] suggested a data structure for a multi-illumination model renderer. That data structure, with minor variations, has been used in this study. In [Ref. 3] a complete derivation of the data is presented, so for the purposes of this study only a brief description is given here. Figure 2.2 illustrates the layout of the data structure as used in this study. This data structure essentially consists of arrays of records layed out in a hierarchical organization. Starting from the highest level it consists of the following: a picture record, an array of light records, an array of objects, an array of subobjects, an array of common part records, an array of polygons, and three arrays for the vertices. Each of these is now examined.

1. Picture

Picture is a single record which contains the view data mentioned earlier.

2. Lights

The lights array is an array of records, with one record for each light source in the scene. Each record contains the light data mentioned above.

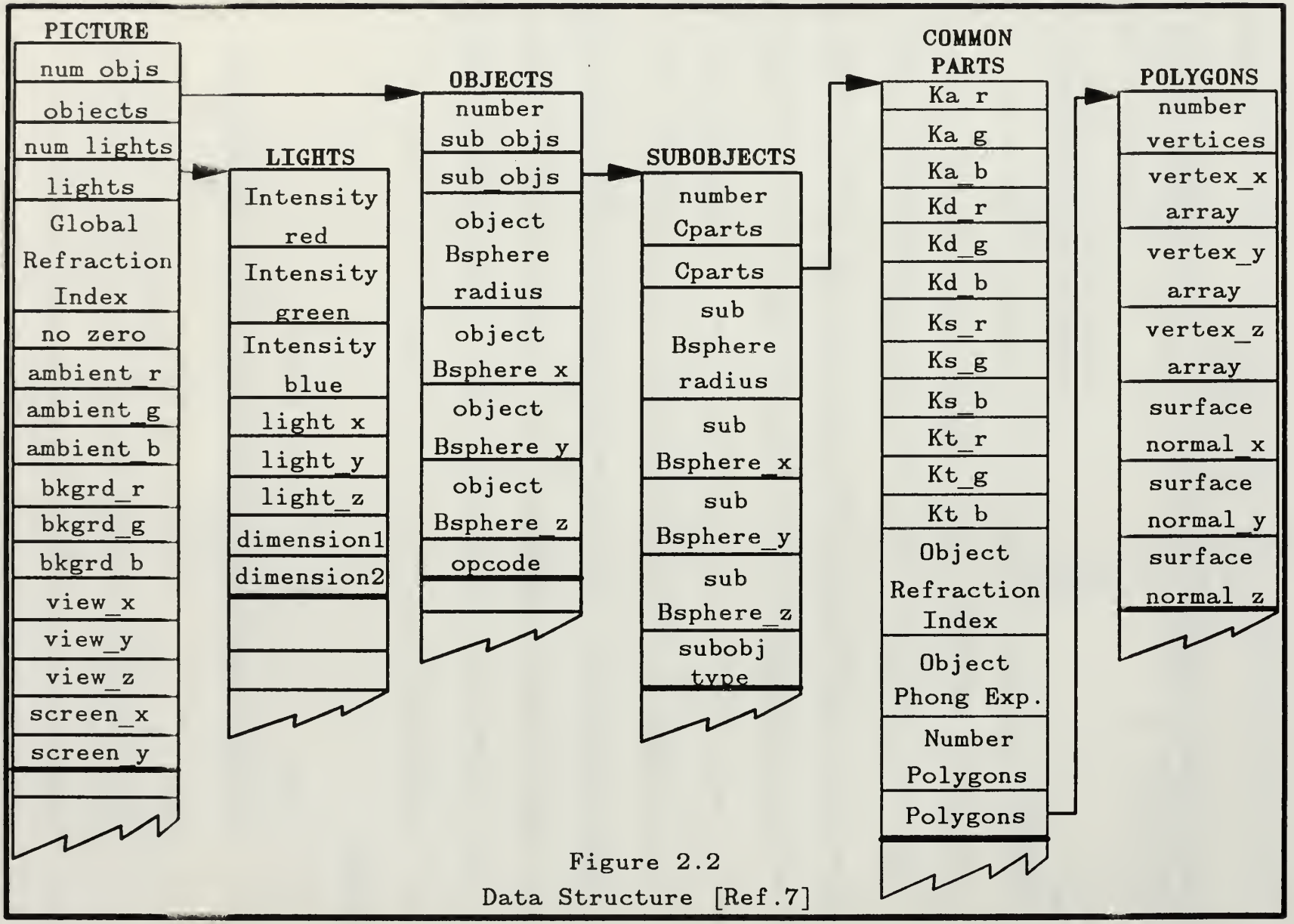

Figure 2.2
Data Structure [Ref.7]

3. Objects

The objects array is an array of records, with one record for each object in the scene. In this study, an object is the highest order item in a scene. Just as the scene is divided up into objects, each with its own bounding volume, so is each object broken down into subobjects, each with its own bounding volume.

4. Subobjects

The subobjects array is an array of records, with each array belonging to one object record. For example Figure 2.3a shows one object, a barbell, that is divided into three subobjects which are: the left weight, the right weight, and the bar. The record layout for this is as illustrated in Figure 2.3c. A subobject is the smallest item in the scene. Each object has at least one subobject. A subobject is composed of polygons or it is a sphere. Using Figure 2.3a as an example again, the left and right weights are spheres and the bar, instead of being a perfect cylinder, is composed of polygons and actually has an octagonal shape, Figure 2.3b. Aside from containing a pointer to the common part record, examined next, and data for its bounding volume, it also contains information on the subobject type. This subobject type field indicates the geometry of the subobject, i.e., it is either a sphere or a polygonal object, which is an object composed of polygons. This information is stored because different intersection routines are used for each object type. Currently a 1 indicates that planar intersection routines should be used and a 0 indicates spherical intersection routines should be used.

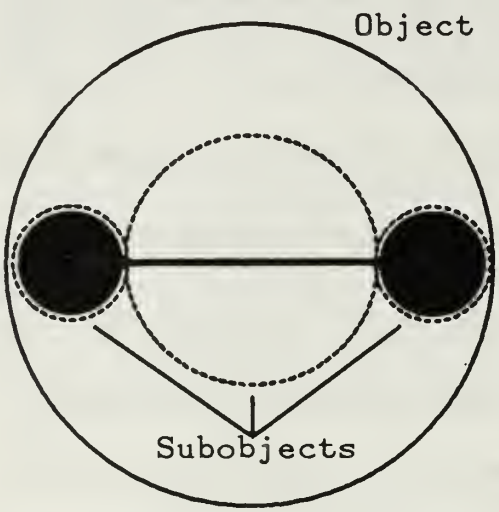

Figure 2.3a
Subobjects [Ref. 7]

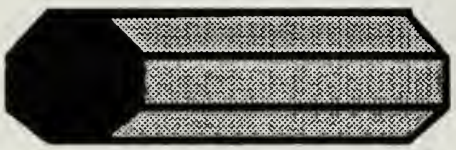

Figure 2.3b
Polygonal Object

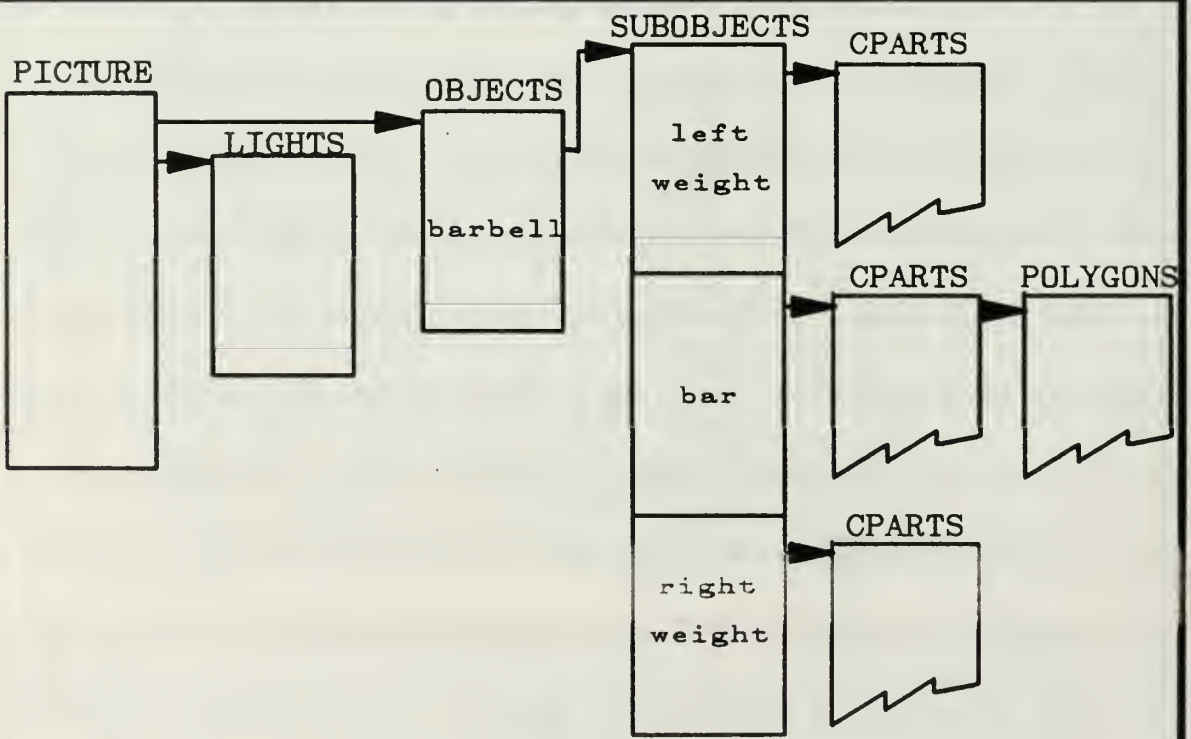

Figure 2.3c
Record Layout for Figure 2.3a

5. Common Parts

A good illustration of common parts is found in an ordinary checkerboard, Figure 2.4a. In this figure one object exists--the checkerboard. This in turn has one subobject, itself. This subobject has two common parts: the white squares and the black squares. Table 2.1 listed the characteristics of an object and it is in the common parts record that these characteristics are stored. Each of these common parts records contains a pointer to an array of polygon records. It is through this arrangement that the polygons inherit the characteristics of the object. Therefore, the common parts array, also called the Cparts array, is an array of records, with one array per subobject, and each common parts record points to its own set of polygons, Figure 2.4b.

6. Polygons

The polygons array, too, is an array of records with one array for each subobject. This is the smallest physical item in the scene and the one against which the actual intersections are determined.

7. Vertex Array

The vertex array is an array of points that define the polygons that compose the subobject.

This data structure as presented by Falby [Ref. 3] proved itself to be both flexible and easy to use. An example of a data base that used this structure and which was used in testing this ray tracer can be seen in Appendix B.

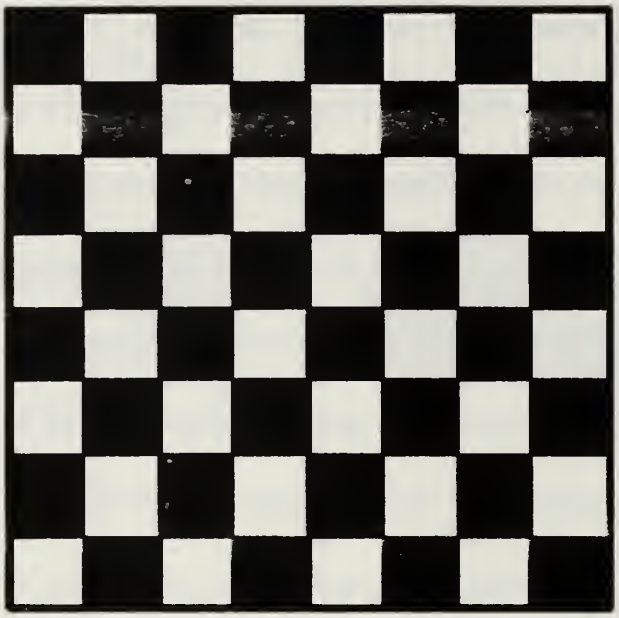

Figure 2.4a - Example of Object with Two Common Parts

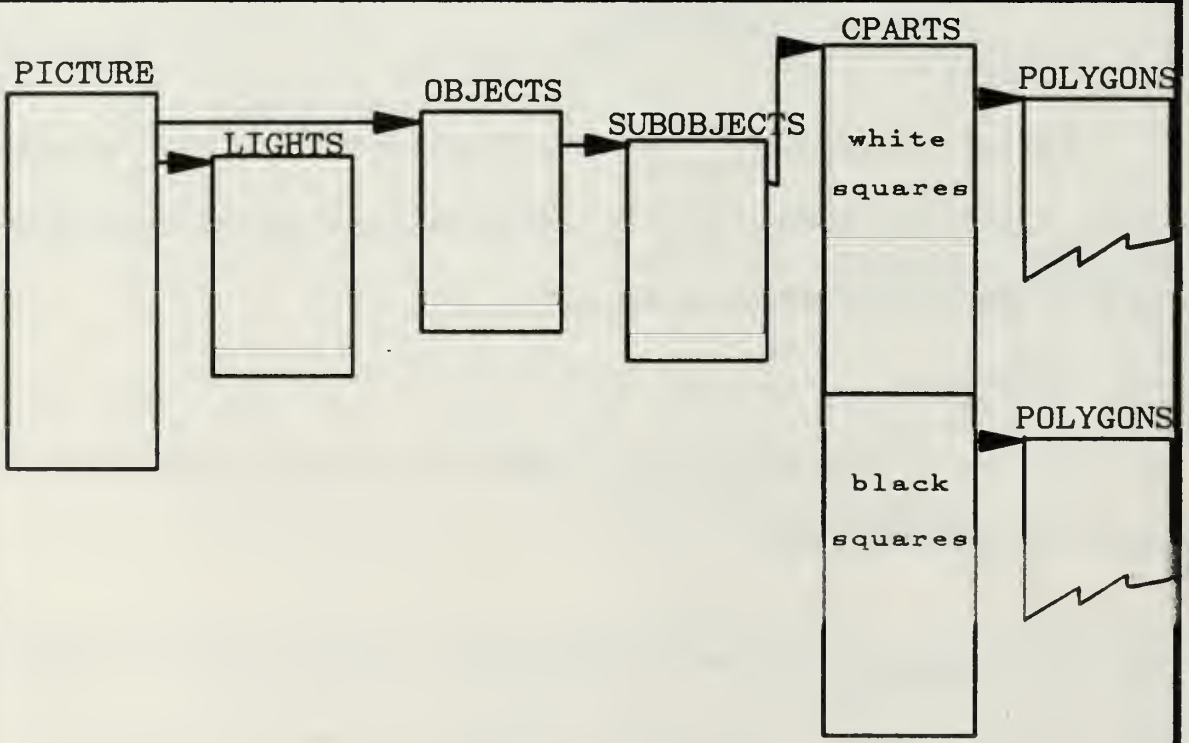

Figure 2.4b - Record Layout of 2.4a

III. RAY TRACING INTERSECTION CONSIDERATIONS

The methodology behind ray tracing is quite simple. However, it does require an understanding of the fundamentals of vector calculus and geometric optics. A discussion of the fundamentals is beyond the scope of this study. An overview of some of the fundamentals is in order.

A. RAY TRACING MECHANICS

By its very definition, ray trącing is simply the tracing, or following, of a ray from its source through space and determining any possible intersections that may occur between it and an object. The natural way to model a ray in order to do this tracing is by using vectors. A vector is not only a precise way to represent a ray but the basic operations on vectors in three space, addition, subtraction, dot product, and cross product provide the tools necessary to determine the intersections. These tools, along with other techniques found in vector calculus and geometric, optics provide the means to deal with the two problems encountered in ray tracing, i.e., the ray direction determination problem ans the intersection problem.

1. The Ray Direction Problem

Solving the ray direction problem is both the first and last step encountered in the ray tracing process. Determining the initial ray from the view

position, usually referred to as the view ray, is the simplest to solve. Every point in a coordinate system can be associated with a ray, and determining the direction of a ray between two points can be solved by using vector subtraction. The last step in the ray tracing process is determining what takes place when a ray intersects an object. This requires the application of the laws of geometric optics. Once a ray strikes an object, either one or two additional rays will be generated. These new rays are referred to as the reflected and refracted rays, Figure 3.1. The three basic laws of reflection and refraction are listed as [Ref. 8: pp: 32-33]

1. The incident, reflected, and transmitted rays all reside in a plane, known as the plane of incidence, which is normal to the surface of the object.
2. The angle of incidence is equal to the angle of reflection $\Theta_i = \Theta_r$.
3. The incident and transmitted ray directions are related by Snells' law:

   $n_i \sin\Theta_i = n_t \sin\Theta_t$.

An illustration of these laws is shown in Figure 3.1. Rogers [Ref. 1: p. 367] provides a method for determining the direction of the reflected and refracted rays. The direction of r, the reflection ray, and p, the refraction ray are given as:

$$r = v' + 2\hat{n}$$

$$p = k_f(\hat{n} + v') - \hat{n}$$

where

$$v' = \frac{v}{|v \cdot \hat{n}|}$$

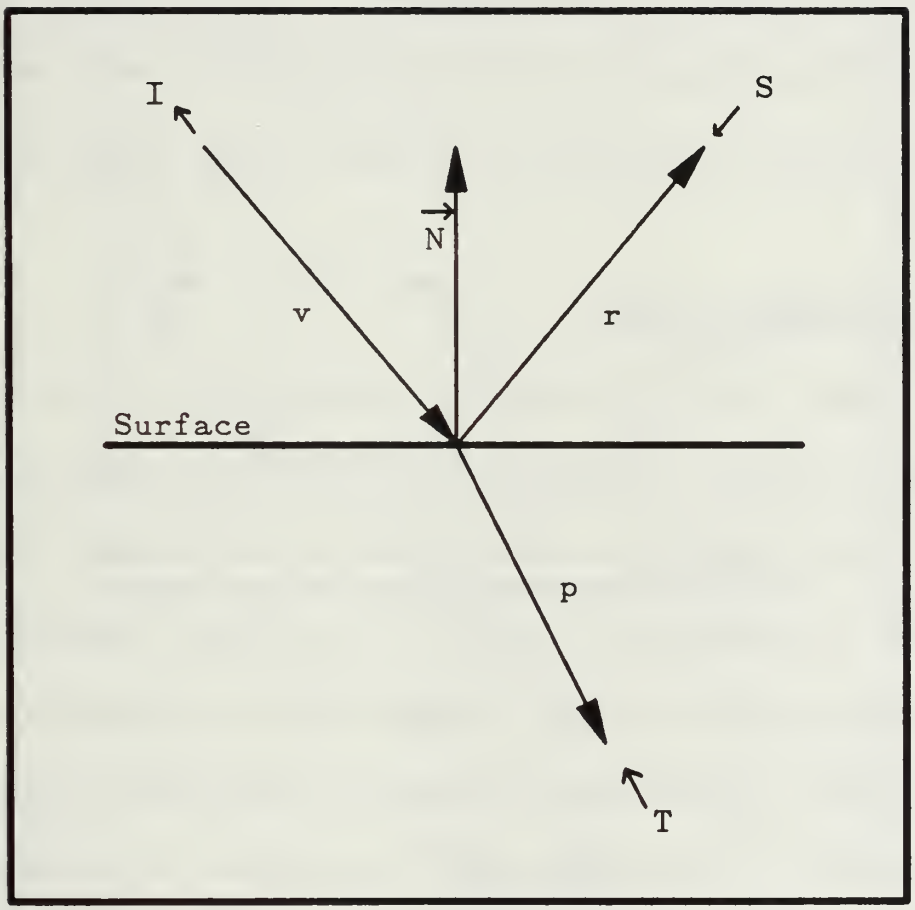

Figure 3.1

Creation of Reflected and Refracted Rays [Ref. 3: p.55]

$$k_f = (k_n^2 |v'|^2 - |v' + \hat{n}|^2)^{\frac{-1}{2}}$$

$$k_\eta = \frac{\eta_2}{\eta_1}$$

Here k_η is the ratio of refractive indices, k_f is the Fresnel coefficient, v is the normal vector in the direction of the incoming ray, v' is the unit normal vector in the direction of the incoming ray, $\hat{n}$ is the unit surface normal, and η_1 and η_2 are the refraction coefficients for mediums the rays pass through. This is illustrated in Figure 3.2.

2. The Intersection Problem

As stated above the intersection computation is the most time consuming part of the ray tracing process. It is not that the process itself is so difficult but because several steps need to be done for each iteration. Two types of intersection computations are required to be performed: determining the intersection between a line and a sphere and determining the intersection between a line and a polygon. The first type is the simplest to solve and is why the sphere is generally used as the bounding volume. The calculation of the intersection point between a line and a sphere involves solving the equation for the line and the sphere simultaneously. The sphere is defined by the equation

$$(x - \alpha)^2 + (y - \beta)^2 + (z - \delta)^2 = r^2 \tag{3.1}$$

where (α, β, δ) is the center point, r is the radius, and (x, y, z) is a point on the sphere. The line is defined by the parametric equations

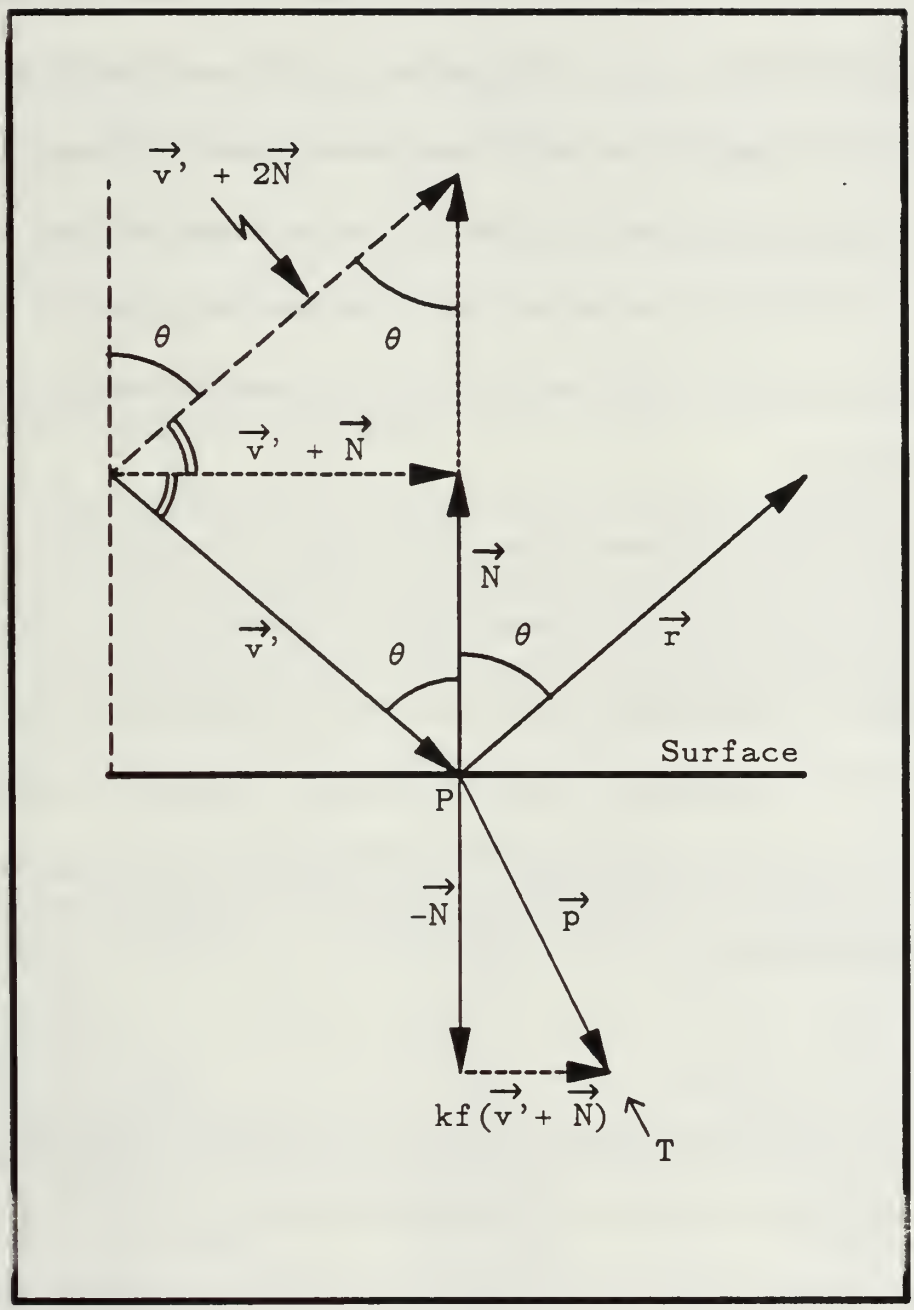

Figure 3.2

Direction of Reflected and Refracted Rays [Ref. 3: p.57]

$$x = at + x_0 \qquad y = bt + y_0 \qquad z = ct + z_0 \tag{3.2}$$

where (x_0, y_0, z_0) is a known point on the line, and a, b, and c are coefficients from the vector $a\vec{i} + b\vec{j} + c\vec{k}$ which is parallel to the line. These equations must first be solved for t. The solution to this provides two answers. First, it indicates whether or not an intersection actually takes place. Second, if there is one, it indicates how many intersections, either one, in the case where the line is tangent to the sphere, or two, in the case where it actually enters the sphere. In the case of two intersection points, a check must be done to determine which is closer to the origin of the ray.

The intersection between a line and a polygon is more involved. This problem is comprised of two parts: determining the intersection point between a line and a plane, and determining whether or not the intersection point lies within the polygon. Like the line-sphere intersection problem, this one also involves the solving of two equations simultaneously. The first of these is the equation for a plane which is defined as

$$Ax + By + Cz = D \tag{3.3}$$

where A, B, C, and D are constants and (x, y, z) is a point on the plane. The second equation used is the parametric equation that defines a line, Eq. 3.2. The solution of this requires first substituting the equations for the line into the equation of the plane. The result is an equation in t, which when solved and substituted back into the equations for the line, provides the intersection point

between the line and the plane. Once this point is calculated, it is then necessary to determine whether or not it lies within the polygon. A general way of solving this is to simply determine the relationship between the intersection point and each edge of the polygon. The point that lies on the inside of each edge also lies within the polygon. If it fails the test for any edge, then it lies outside the polygon. The drawback to this approach, is that it only works for convex polygons. In this study we assume all polygons are convex. [Ref. 9].

B. THE RAY DATA STRUCTURE

Rogers [Ref. 1] suggests a data structure for a ray in a ray tracer. It is that data structure which is used in this study. Table 3.1 lists the data used to model each ray. We examine each item of this structure as adapted from Rogers [Ref. 1: p. 373].

TABLE 3.1 - RAY DATA

FIELD NAME	VARIABLE NAME	VALUE
Ray Type	type	enumerated or coded
Ray Origin	x, y, z	real
Ray Vector	x, y, z	real
Source Ray Type	Stype	enumerated or coded
Intersection Flag	flag	boolean or coded
Object Index	obj_idx	integer
Subobject Index	subobj_idx	integer
Common Part Index	Cpart_idx	integer
Polygon Index	polygon_idx	integer
Intersection Point	x, y, z	real
Distance	d	real
Transmitted Intensity	I_t	real
Specular Intensity	I_s	real

1. Ray Type

The ray type field identifies a ray as either a view ray, a reflected ray, or a refracted ray. The values put in this field are generally of an enumerated type and consist of *reflected, refracted, view,* and *none.*

2. Ray Origin

The ray origin field contains the point that identifies the position from which the ray originated. For instance, if it is the view ray, its point of origin is the view position. If it is a reflected or refracted ray, its origin is the intersection point that it originated from.

3. Ray Vector

The ray vector field contains the vector heading of the ray.

4. Source Ray Type

The source ray type field contains the ray type of the source ray for this particular ray. For instance, a view ray does not have any source ray as it is the starting ray for the process. Hence, *none* is in the type field. If the view ray intersects an object and both a reflected and refracted ray are generated, then the source ray for both of them is the view ray. Likewise if the reflected or refracted ray hits an object and creates further rays then it becomes the source ray for those rays it creates.

5. Intersection Flag

Originally the intersection flag is set to false and it is only set to true when there is an intersection between this ray and an object.

6. Object Index

The object index provides an index into the array of object records making it possible to select any object easily.

7. Subobject Index

The subobject index provides an index into the array of subobject records and helps uniquely identify each subobject.

8. Common Part Index

The common part index provides the index into the array of common part records uniquely identifying each common part record.

9. Polygon Index

The polygon index provides the index into the array of polygon records and uniquely identifies each polygon.

10. Intersection Point

The intersection point field holds the position of the intersection point between the current ray and an object.

11. Distance

The distance field contains the distance between the current ray's point of origin and its point of intersection.

12. Transmitted Intensity

The transmitted intensity field contains the red, green, and blue intensity values, in a range between 0 and 1, of the light that is incoming along any refracted ray that this ray produces.

13. Specular Intensity

The specular intensity field contains the red, green, and blue intensity values, in a range between 0 and 1, of the light that is incoming along any specular ray that this ray produces.

C. INTERSECTION METHODOLOGY

1. Intersecting a Planar Polygon

a. Generating the Initial Ray

The generation of the initial or view ray is shown in Figure 3.3. This ray, $p - v$, is calculated using vector subtraction. The two vectors used are the ones associated with the points for the view position, v, and the pixel through which the ray passes, p. The ray associated with the view position is to be subtracted from the ray associated with the pixel.

b. Intersecting the Bounding Volumes

After each ray is generated, each object in the object list is checked, one at a time, to determine whether or not the ray strikes any of the bounding containers of the objects in the scene. In our implementation, the bounding container is the sphere, which is reduced to a bounding circle, C, see Figure 3.4. This circle's radius is the same as the radius of the bounding sphere, S, and it lies on the plane, P, which contains the center point, q, of the bounding sphere. The inverse of the incoming ray, $\vec{v}$, is the surface normal of this plane. It is not necessary to determine where on the bounding sphere a ray hits since at this stage

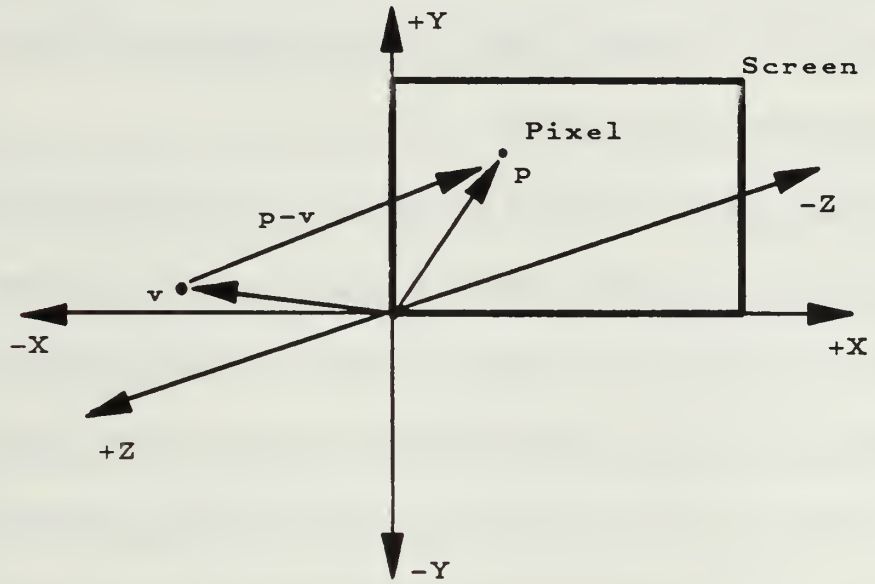

Figure 3.3 - Determining the Ray View

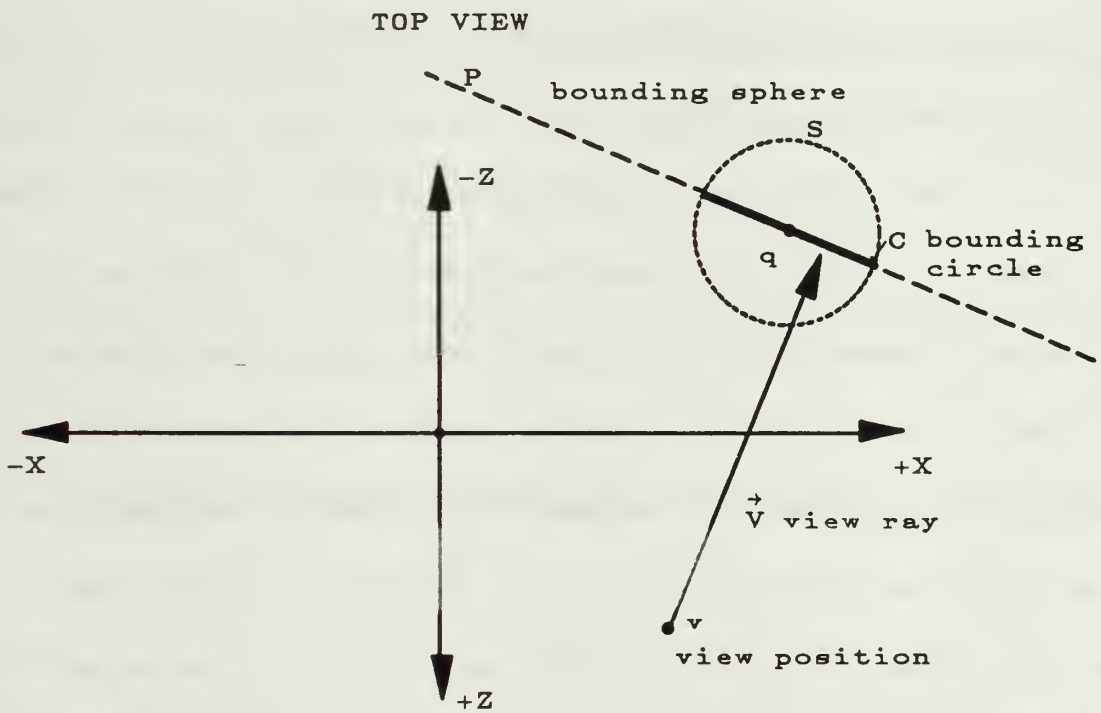

Figure 3.4 - The Bounding Circle

we are only interested in just finding out whether or not it hits it. Because we are not concerned with where the ray strikes the bounding sphere, a bounding circle is used. Determining the intersection with a circle requires less work than determining the intersection with a sphere.

The first step in tracing the ray begins by taking each object and constructing a bounding circle. Once this circle is constructed, the intersection point of the view ray with it is calculated. Then the distance between the intersection point and the center point is determined and compared to the radius of the bounding circle to see if it falls within the circle. If it does fall within the circle, this indicates that the ray intersected the bounding volume. This process then needs to be repeated for each object. If an object's bounding circle is hit, this process must then be repeated for each subobject of that object.

c. Intersecting the Polygon

Once a particular subobject is identified by the bounding volume tests, the common parts list is processed, one record at a time. Each one of these common parts records contains a pointer to those polygons that make up the object being rendered. This list of polygon records is then processed after the bounding volume processing is completed. The entire list needs only to be processed until an intersection is found. The processing involved in this is the most computationally expensive part of the entire ray tracing algorithm. This computation consists of three steps: determining the orientation of the polygon, calculating the intersection point between the ray and the plane that contains the

polygon, and determining whether or not the intersection point lies on the polygon.

(1) Establishing the Orientation. First, each polygon needs to be checked for a correct orientation. This is a straightforward step carried out by calculating the angle between the surface normal of that polygon and the inverse of the view ray. If it is 90 degrees or greater, it is facing in the wrong direction to be intersected. If it is less than 90 degrees, the next step is to determine whether the ray intersects the particular plane that that particular polygon lies on. The first step in doing this is to determine the equation for the plane in which that polygon lies.

(2) Intersecting a Plane. If the correct orientation exists for a polygon to be intersected, the next step is to define the plane containing the polygon of interest. The equation for a plane was given earlier as

$$Ax + By + Cz = D$$

where A, B, C, and D are constants and can be calculated by the following equation.

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$

$$B = z_1(x_2 - x_3) - z_2(x_3 - x_1) - z_3(x_1 - x_2)$$

$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

$$D = -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1)$$

where (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3) are points of the vertices of the

polygon. Once the equation for the plane is known, it must then be solved simultaneously with the equation of the line representing the ray.

(3) Location of a point with respect to a polygon. Once the intersection point is determined, it needs to be checked to see whether or not it lies within the polygon. This is the most computationally expensive part of the process. This process, see Figure 3.5, requires that a plane, P, called the bounding plane, containing two vertices, for example A and B, of an edge and another arbitrary point, K, not on the polygon or its plane, be constructed. This must be done for each edge. Once the bounding plane is constructed, the point in question must then be checked to see whether or not it lies to the polygon's side of the plane that now contains the edge of the polygon. This is done by plugging in another vertex of the polygon into the equation for the plane that was just constructed, and then plugging in the intersection point. If the results from these two equations have the same relationship, i.e. if the sign is the same, then the intersection point lies on the polygon side of the bounding plane. This check must then be repeated for each edge of the polygon. In order for the intersection point to lie on the face itself, it must be found to lie on the polygon's side of each edge. At this point, it is determined whether or not the ray strikes the object. If the ray does strike the object, the pointers to this polygon must be stored in the ray structure along with the intersection point. This information is needed later on when determining the direction of reflected and refracted rays as well as the intensity of the light that is reflected from this position.

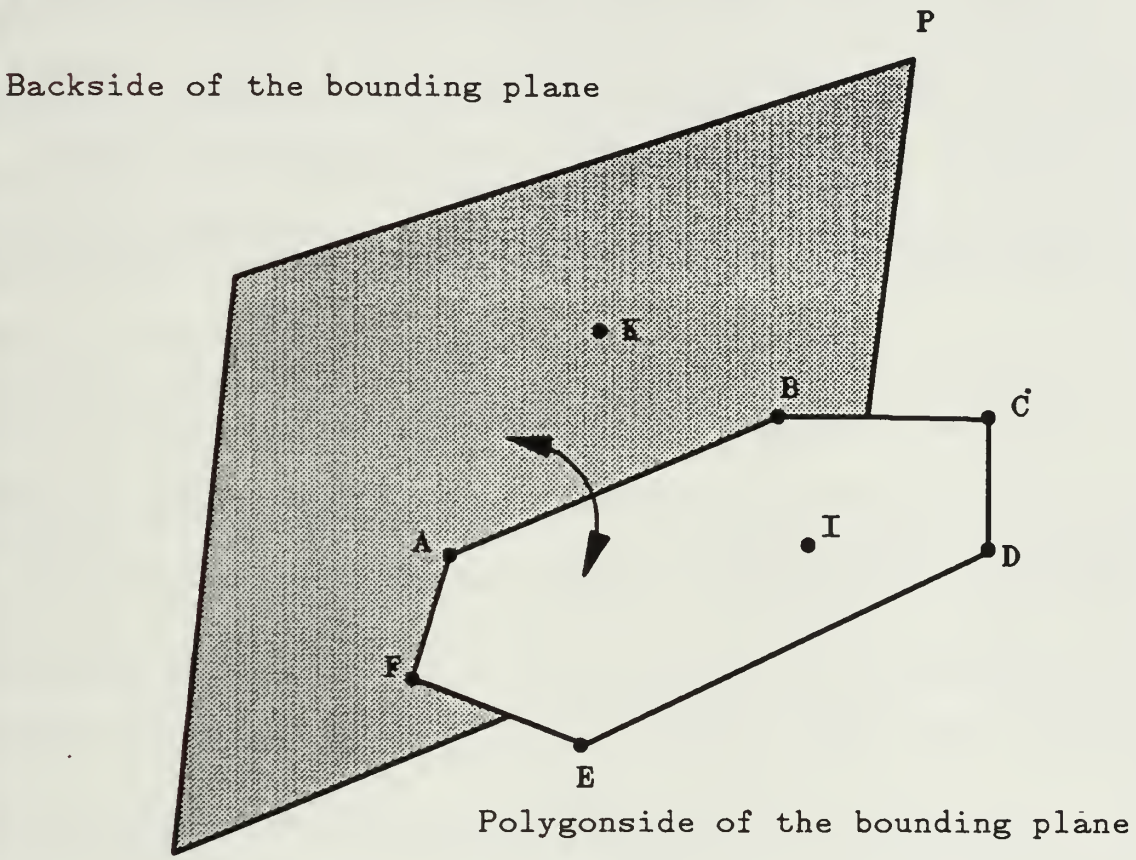

Figure 3.5
Location of a point with Respect to a Polygon [Ref. 9]

2. Intersection of a Sphere

The sphere is the easiest object to work with in a ray tracer. Since the sphere can act as its own bounding volume, the center point and radius are already available in the subobject record. This eliminates the need for the polygon array. The center point and radius are the only information necessary to model a sphere for a ray tracer. Determining the intersection of a line and a sphere is nothing more than the simultaneous solving of their equations for the variable **t** . The solution to this gives a quadratic equation in **t** which can then easily be solved.

IV. THE INTENSITY PROBLEM

One of the strong points of the ray tracing algorithm is that a global illumination model can easily be integrated into it. In fact, ray tracing and global illumintion models seem to naturally complement each other. A global illumination model takes into account all of the light sources in the scene in calculating the intensity at each point. This means taking into account the ambient light that exists in a scene, light that comes directly from a particular light source(s), and light that is reflected off an object(s). It also includes the coefficients necessary to model the way an object reacts with light. A great deal of work has been done in this area. The most notable model is the Whitted illumination model, and it is the one that has been implemented here [Ref. 1: pp. 365-366]. The Whitted algorithm is based on the three models shown in Figures 4.1, 4.2, and 4.3. These models will now be examined more closely.*

A. LOCAL ILLUMINATION MODELS

1. Diffuse Reflection Model

The first of these models is a perfect diffuser. Such a model is governed by Lambert's cosine law. This law states that the intensity of light reflected from a perfect diffuser is proportional to the cosine of the angle between the light

*The contents of this chapter are close adaptations from Rogers [Ref. 1] and Falby [Ref. 3].

direction and the normal to the surface. This can be expressed mathematically as

$$I = I_l k_d \cos\Theta \qquad 0 \leqslant \Theta \leqslant \frac{\Pi}{2}$$

where I is the reflected intensity, I_l is the incident intensity from a point light source, k_d is the diffuse reflection constant, unique to each object, and Θ is the angle between the light direction and the surface normal, see Figure 4.1. Since the diffuse reflection coefficient k_d varies from material to material and is also a function of the wavelength of the light, it is often easier to just assume it a constant for simple illumination models. [Ref. 2: p. 312]

2. Specular Reflection Model

The second model illustrates the characteristics of specular reflection which is directional, unlike diffuse reflection. This means that the greatest intensity of the specularly reflected light can only be seen if the view angle coincides with the reflection angle, Figure 4.2. The further off the viewing angle is from the reflection angle, the dimmer the intensity becomes. Because of the complex physical characteristics of specularly reflected light, an empirical model due to Bui-Tuong Phong is usually used for simple illumination models [Ref. 10]. This is expressed mathematically as

$$I_s = I_l w(\Theta_i, \lambda) \cos^n \alpha$$

where $w(\Theta_i, \lambda)$, the reflection curve, gives the ratio of the specularly reflected light to the incident light as a function of the incidence angle Θ_i and the wavelength λ. Because $w(\Theta_i, \lambda)$ is such a complex function, it is frequently

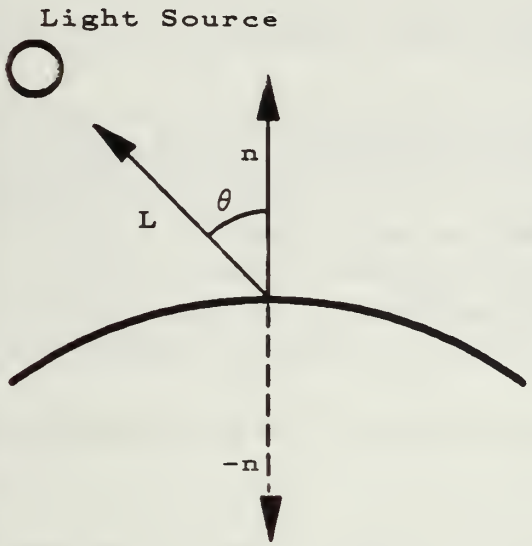

Figure 4.1
Diffuse Reflection
[Ref. 1: p.312]

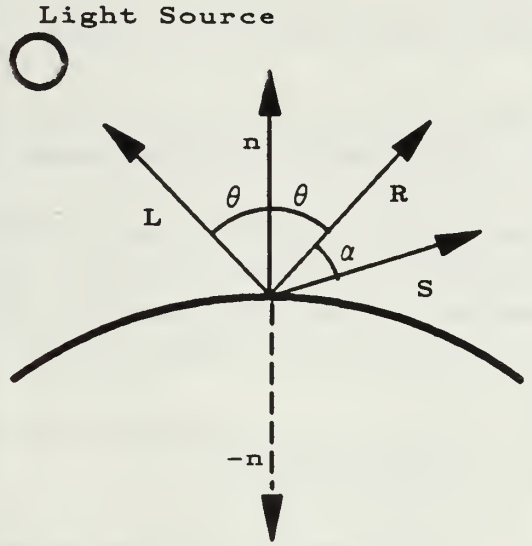

Figure 4.2
Specular Reflection
[Ref. 1: p.314]

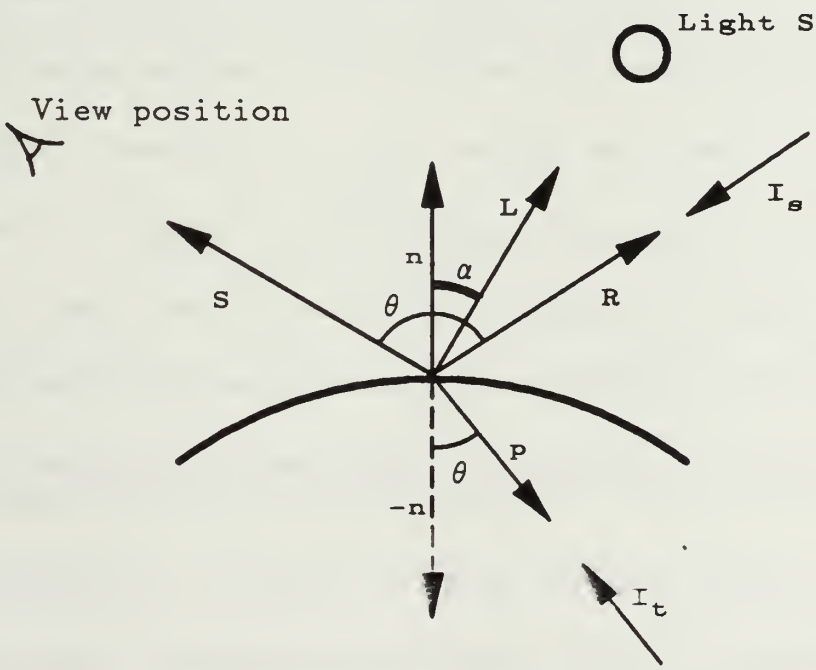

Figure 4.3
Global Illumination Model

replaced by an aesthetically or experimentally determined constant k_s which then yields

$$I_s = I_l k_s \cos^n \alpha$$

Also, n is a power that approximates the spatial distribution of the specularly reflected light. Typically a value of 200 for n is used to model a very shiny surface and a value of 10 is used for a dull surface [Ref. 3: p. 72]. [Ref. 1: pp. 313-315]

3. Combined Model

If just point sources are assumed, as in the two models just discussed, any object not receiving light directly from the source appears black. In order to properly render a scene, it is also necessary to take into account ambient light, the light that is reflected off other surfaces. Including a model for ambient light into the intensity calculations is not feasible. Ambient light represents a distributed light source and as such is a very complex function. Therefore it is treated as a constant diffuse term and linearly combined with the other terms. Also not included in the above model is the effect that distance has on light. It is well known that the farther away an object or light source is, the dimmer it gets. The actual formula to produce that proper attenuation is $\frac{L}{d^2}$, where the intensity of light decreases as the square of the distance from the source increases. However it has been shown that linear attenuation can actually produce more realistic results. With these two additions the complete model now looks like:

$$I = I_a k_a + \frac{I_l}{d + K}(k_d \cos\Theta + k_s \cos^n \alpha)$$

where I_a is the incident ambient light intensity, k_a is the ambient diffuse reflection constant and K is an arbitrary constant that can be used to adjust the results. [Ref. 1: p. 313]

The complete model just presented can now be modified to better fit in with a ray tracer. Recalling the formula for the dot product of two vectors allows writing the $\cos\Theta$ as $\hat{n}\cdot\hat{L}$ and writing $\cos\alpha$ as $\hat{R}\cdot\hat{S}$ which gives us:

$$I = I_a k_a + \frac{I_l}{d + K}[k_d(\hat{n}\cdot\hat{L}) + k_s(\hat{R}\cdot\hat{S})^n]$$

So far we have only examined the case where just one light source is present. If there are several light sources, the effects are added linearly, and the equation now becomes:

$$I = I_a k_a + \sum_{j=1}^{m} \frac{L_{l_j}}{d + K}[k_d(\hat{n}\cdot\hat{L}_j) + k_s(\hat{R}_j\cdot\hat{S})^n]$$

This then is the complete local illumination model. [Ref. 1: pp. 312-316]

B. GLOBAL ILLUMINATION MODEL

The complete local illumination model just presented forms the basis for the algorithm that was implemented for this study [Ref. 1: pp. 363-378] see Figure 4.3.

$$I = k_a I_a + k_d \sum_j I_{l_j}\left(\hat{\mathbf{n}} \cdot \hat{\mathbf{L}}_j\right) + k_s \sum_j I_{l_j}\left(\hat{\mathbf{S}} \cdot \hat{\mathbf{R}}_j\right)^n + k_s I_s + k_t I_t$$

In the above equation k_a, k_d, k_s, and k_t are the ambient, diffuse, specular reflection, and transmission coefficients, all of which have values between 0 and 1. I_a, I_s, I_t, and I_{l_j} are the intensities of the ambient light, the specularly reflected light, the transmitted light, and the light directly from a light source. These also hold values between 0 and 1. The remaining variables $\hat{\mathbf{n}}$, $\hat{\mathbf{L}}_j$, $\hat{\mathbf{S}}$, and $\hat{\mathbf{R}}_j$ are the surface normal at the intersection point, the direction of the jth light source, the local sight vector, and the local reflection vector from the jth light source. A careful comparison between this model and the complete local illumination model reveals that the only new terms are the $I_s$and I_t terms. These are the terms used to account for the light that comes in along the reflected and refracted rays that originated at this point. I_s holds the intensity for the reflected ray and I_t holds the intensity for the refracted ray. These two values in turn are calculated using the exact same model. For the last intersection point in the scene, the one whose reflected and refracted rays do not intersect anything, I_s and I_t are set to 0. The k_s and k_t terms are coefficients included to better model the way this object reacts with the light incoming along the reflected and refracted rays.

This then is the complete global illumination model used in this study. It is the simplicity of this algorithm that makes it so easy to understand and implement. In essence, it is saying that the output intensity is nothing more than

a sum of all possible light sources with the coefficients determining the intensity of light that comes from a particular object.

V. RAY TRACING ALGORITHM

Rogers suggested an algorithm for ray tracing [Ref. 1: pp. 374-377]. It is that algorithm that has been the basis for this study. The following is a description of that algorithm as it has been implemented here.

A. TRACING THE RAYS

To begin the ray tracing process, the first thing done is the determination of the direction of the view ray.* The ray data, mentioned in chapter III, is then initialized. After the view ray is generated and the ray data initialized, the ray, which is represented by this ray data, is pushed onto the stack, which is used to implement the ray tracing tree. The process then moves into the actual ray tracing loop.

Once in the loop, the stack is checked to see if it is empty. If it is not empty, the stack is then popped to access the ray information. The first thing checked is whether or not the intersection flag is set. If it is, that indicates that that particular ray has already been terminated (by hitting an object), and the process of determining the intensities begins.

*Each ray, is modeled as a vector, and is converted to a unit vector immediately after its determination.

If the intersection flag is not set, a new ray has been generated, either a new view ray or a new reflected or refracted ray. All of these rays are grouped under the more general title of a shooting ray. At this stage, the ray must be sent through the intersection procedures to determine whether or not it intersects any object. The intersection routines start at the highest level of the picture and step through the linked list of objects, subobjects, and common parts to the actual polygons.

At the top level of the intersection routines, each one of the objects is checked. First, they are checked to see if there is an intersection with the object's bounding circle. If there is an intersection, the distance between the intersection point and the origin of the ray is calculated and placed in the ray data. Second, a check is done to insure that the objects lie in front of the ray's origin. In front refers to those objects that lie in the direction of the shooting ray from its point of origin. Since the ray is being modeled by a vector, which only indicates direction, every object along the line described by the vector and the origin of the ray is considered, see Figure 5.1. The way to test for this is to generate a test vector between the origin of the ray and any intersection points of the shooting ray, eliminating the intersection points that lie in the opposite direction from the shooting ray.

As each object is processed, the objects whose bounding volumes are intersected by the shooting ray are processed further to see if any of their subobjects are also intersected. The same basic procedure for finding an

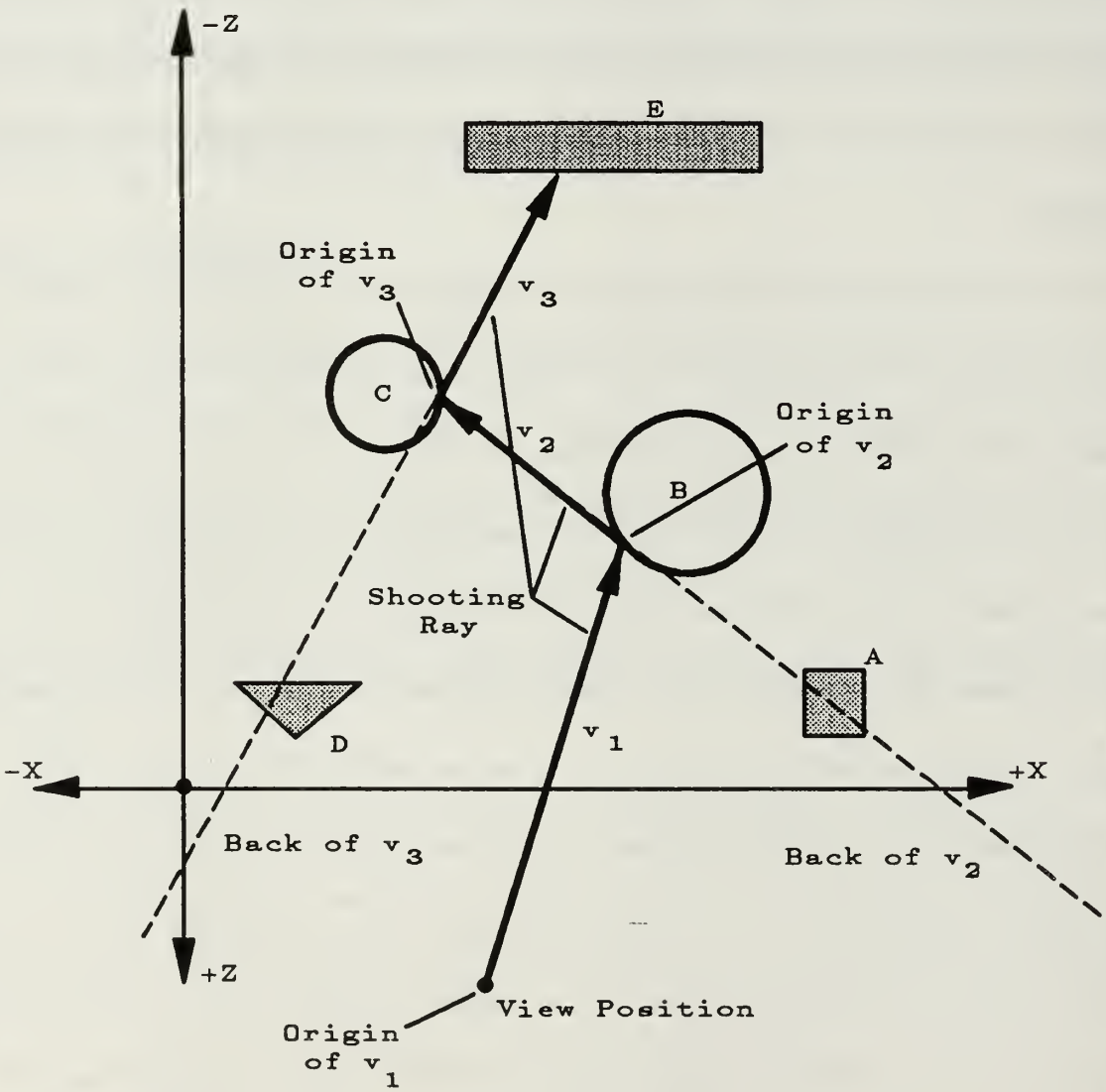

Figure 5.1 - Object Locations

intersection is used here as is used for finding an intersection for the objects. If the object and the subobject are one and the same, this check need not be done.

For each subobject whose bounding volume is intersected, the next step is to check the type of geometric object, i.e., a sphere or a polygonal object. This needs to be done in order to determine which intersection routines are needed. In the case of a sphere, it becomes just a matter of solving two simultaneous equations, discussed earlier--one for a line, the other for the sphere. When dealing with a polygonal object, it becomes more complicated because each face of the polygon must be checked. The first thing that needs to be done is to check the orientation of the face in question. When the polygon does have the right orientation, the intersection process continues. The first step is to determine the equation of the plane that contains the polygon. This plane is calculated from any three vertices of the face in question. Once the plane has been established, the intersection point between it and the shooting ray is calculated. With the intersection point thus established, the next step is to determine whether or not the point lies on the face of the polygon. If the intersection point is found to lie on the face of the polygon, then this point is saved and placed in the ray data.

If no intersection is found and the shooting ray is either a reflecting or refracting ray, then it is discarded and the intensity calculation process begins. If the shooting ray is a view ray, then the intensity is set to the background intensity which is then displayed and the next ray is generated.

When there is an intersection, the first thing to be checked is whether or not there is enough room on the stack. Since the stack holds only part of the ray tree at any one time, it need only be long enough to contain the longest anticipated branch. A particular branch of the ray tree is terminated when both the reflected and refracted rays at an object intersection leave the scene or when the available stack length is exceeded. When both rays leave the scene, their contribution to the illumination at the source ray is zero. When the available stack length is exceeded, the algorithm calculates the illumination at the source ray using only the ambient, diffuse, and specular reflection components at the source ray intersection. An extension algorithm is given in Rogers whereby the algorithm can be extended one additional depth in the tree without exceeding the maximum stack depth. However, the implementation of that was not necessary here. When the stack does get full, it becomes a matter of calculating the intensity at that intersection point and setting the appropriate value I_t or I_s in the source ray. [Ref. 1: p. 372]

When the stack is not full, then the distance between the source point of the shooting ray and the intersection point is determined and placed in the ray data. The ray is then placed back on the stack. Once that is done, any reflecting and/or refracting rays emanating from the intersection point are determined, their ray data initialized, and then placed on the stack as new rays with the reflecting ray being placed on first. It is important to keep this order of rays in mind because it is necessary to know the number of rays to pop when setting the intensities of the

source rays. With these new rays in place on the stack, the program returns to the beginning of the ray tracing cycle. In the absence of reflecting or refracting rays, then the first ray popped is the view ray. Now since this ray already has its intersection flag set, the intensity at the point of its intersection is calculated and displayed. If the ray popped is a reflective or refractive ray, then it becomes the new shooting ray. This cycle continues until either no more reflecting or refracting rays are produced or until the stack becomes full. This process is summarized by the pseudocode description in Figure 5.2.

B. DETERMINING THE INTENSITY

If a ray's intersection flag is found set at the beginning of the ray tracing process, that ray is sent into the intensity processing part of the ray tracer. The first step in this process is to take the ray data and to determine the intensity at that ray's intersection point. This intensity is then divided by the distance between the ray's point of origin and its intersection point in order to properly attenuate it. This process is done for each of the primary colors -- red, green, and blue. If a view ray is being considered, that means that it was the last ray on the stack. Therefore, it is the final intensity to be calculated and hence is the actual one displayed. If it is a refracted ray, then the intensity just calculated becomes the I_t intensity in the source ray for the ray currently being examined. If it is a reflected ray, then the intensity just calculated becomes the I_s intensity in the source ray for the ray currently being examined. The stack is set up so that the

```
READ IN DATA FILE

for Y := 1 to MAX_ROWS do
  for X := 1 to MAX_COLUMNS do
    INITIALIZE VIEW RAY
    PUSH RAY ONTO STACK
    repeat
      POP RAY FROM STACK
      if INTERSECTION FLAG SET then
        CALCULATE INTENSITY
      else
        CHECK FOR INTERSECTION
        if INTERSECTION FLAG SET then
          if STACK EXCEEDED then
            CALCULATE INTENSITY
          else
            PUSH RAY BACK ON STACK
            CALCULATE REFLECTED RAY
            CALCULATE REFRACTED RAY
            if REFLECTED RAY EXISTS then
              INITIALIZE REFLECTED RAY
              PUSH REFLECTED RAY ON STACK
            end if
            if REFRACTED RAY EXISTS then
              INITIALIZE REFRACTED RAY
              PUSH REFRACTED RAY ON STACK
            end if
          end else
        else
          if (CURRENT RAY TYPE = VIEW RAY) then
            SET INTENSITY TO BACKGROUND COLOR
        end else
    until STACK EMPTY
    DISPLAY PIXEL
  end FOR-X
end FOR-Y
```

Figure 5.2 - Pseudocode Description of the Ray Tracing Process

source ray is on the bottom of the ray tuple with the reflected ray above it and the refracted ray above that. In order to set these values, it is necessary to pop the stack to gain access to the source ray. Once these values are set in the source ray, the program returns to the beginning of the loop and pops the next ray off the stack discarding the last ray as it is no longer needed. [Ref. 1: p. 377]

The intensity algorithm [Ref. 1: p. 377] although very simple in design, depending on the number of light sources in the picture, can also become a time intensive part of the ray tracing program. The entire ray data set, listed in Table 3.1, is sent into this routine as well as the pointers to the object and light source list. As in the intersection routine, rays are generated here. In Rogers, they are referred to as shadow feelers and the same term is used in this study. These shadow feelers are those rays represented as the vectors from the point of intersection to the light source, see Figure 5.3. They are used to determine the intensity contributed to that point from that light source. Once these rays are generated, they also pass through the intersection routines in order to determine which objects, if any, the light rays pass through en route to the intersection point. The first test that must be done is to determine if any of the objects passed through are opaque. If any are opaque. then no light reaches the intersection point from that light source. That point is then considered to be lying in deep shadow with respect to that light source. If none of the objects intersected by the light ray is opaque, then the light intensity needs to be attenuated accordingly for each of them. This attenuation entails multiplying the intensity at each point by

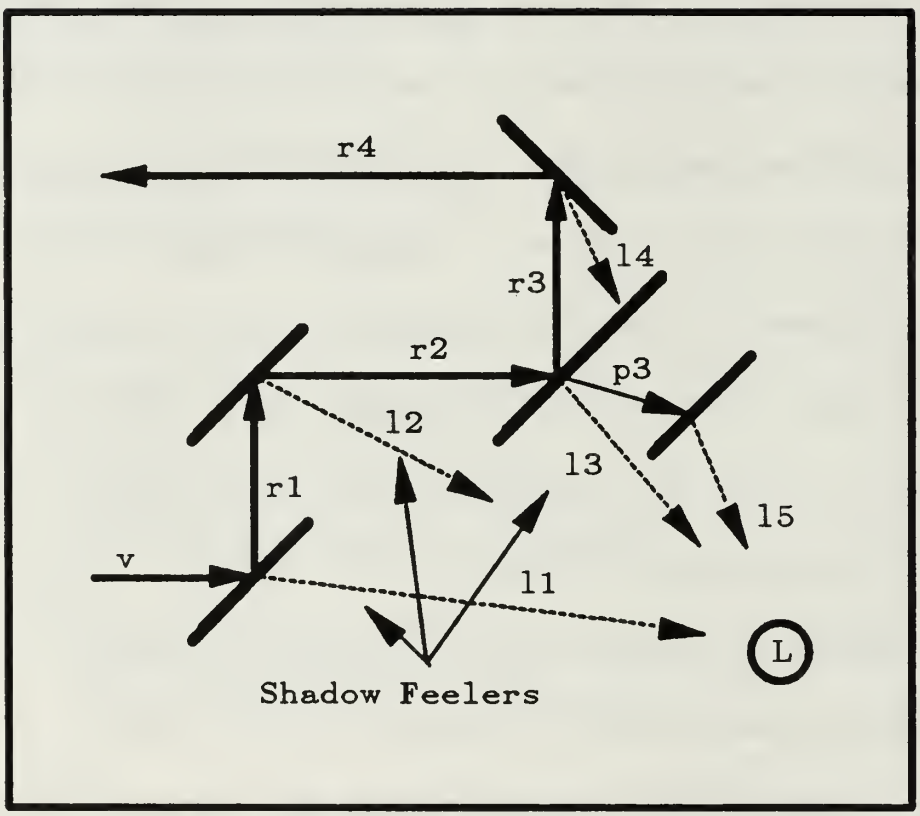

Figure 5.3

Shadow Feelers [Ref. 3: p. 64]

the transmission coefficient of the object. This process then needs to be repeated for each light source. A running total of the intensities is maintained to be included in the final calculation. The calculation takes into account the ambient light and the light that comes in along the reflection and refraction rays. This process then produces the final intensity. If the input ray type is a view ray then it is displayed. If it is a reflected ray, it becomes the I_s value in the source ray. If it is a refracted ray, it becomes the I_t value in the source ray. [Ref. 1: pp. 376-377]

VI. IMPLEMENTATION

The prototype was written in BORLAND's Turbo Pascal and implemented on an IBM AT clone. The program is 2500 lines long and takes two hours to generate a scene of 200x200 pixels. The scenes generated on the AT were then displayed using the RGB monitor of a Silicon Graphics IRIS 2400 graphics workstation.

The main focus of this study was to develop a prototype ray tracer, which by itself is just a hidden surface removal technique. The secondary consideration was to integrate an illumination model into the ray tracer. Because of this focus, more time was spent examining the ray tracing algorithm than any of the global illumination models that could have been integrated along with it.

The top three scenes in Figure 6.1 tested the ability of the prototype to perform as a hidden surface remover. The program proved successful in this area. For these scenes, a stub was used in place of the illumination model and each scene was lit using only ambient light. From left to right the scenes show: An unobstructed view of the three objects, described in a later section; the cube and sphere suspended above the floor but with the cube partially blocking the sphere; the cube and the sphere sunk part way into the floor with the cube still in front.*

*These were the only scenes generated to test for hidden surface removal. The remainder of the tests were done trying to integrate a global illumination model.

The testing of the global illumination model integrated into the ray tracer unveiled some problems. The two bottom scenes in Figure 6.1 are representative of the successful results. The first problem discovered was that attenuating the intensity by the full distance between the origin of the view ray and its intersection point produced totally blackened objects. The results shown in Figure 6.1, the bottom row, were obtained by either dividing the intensity by two, see the scene on the left, or by not dividing it at all, see the scene on the bottom right of the object. The second problem can be clearly seen by the black line that runs up the center of the floor on the bottom right scene. This result along with those test that generated shadows, not shown, indicated that the intensities for the floor were reversed. The black line, clearly seen in color Figure 5 and just vaguely visible in color Figure 4, is actually specular reflection and should be much brighter than the rest of the floor. In those scenes where shadows were generated the shadows were also brighter than the rest of the floor--just the reverse of what it should have been.

A. INPUT

The test data used in this study produced the scenes shown in Figure 6.1. This test data was in the form of a sequential file with the data structure outlined in Figure 2.2. The data had in it one picture record, one light in the lights array, and three objects in the objects array. The first object is a cube which contains one subobject. This subobject has one common part. The common parts record

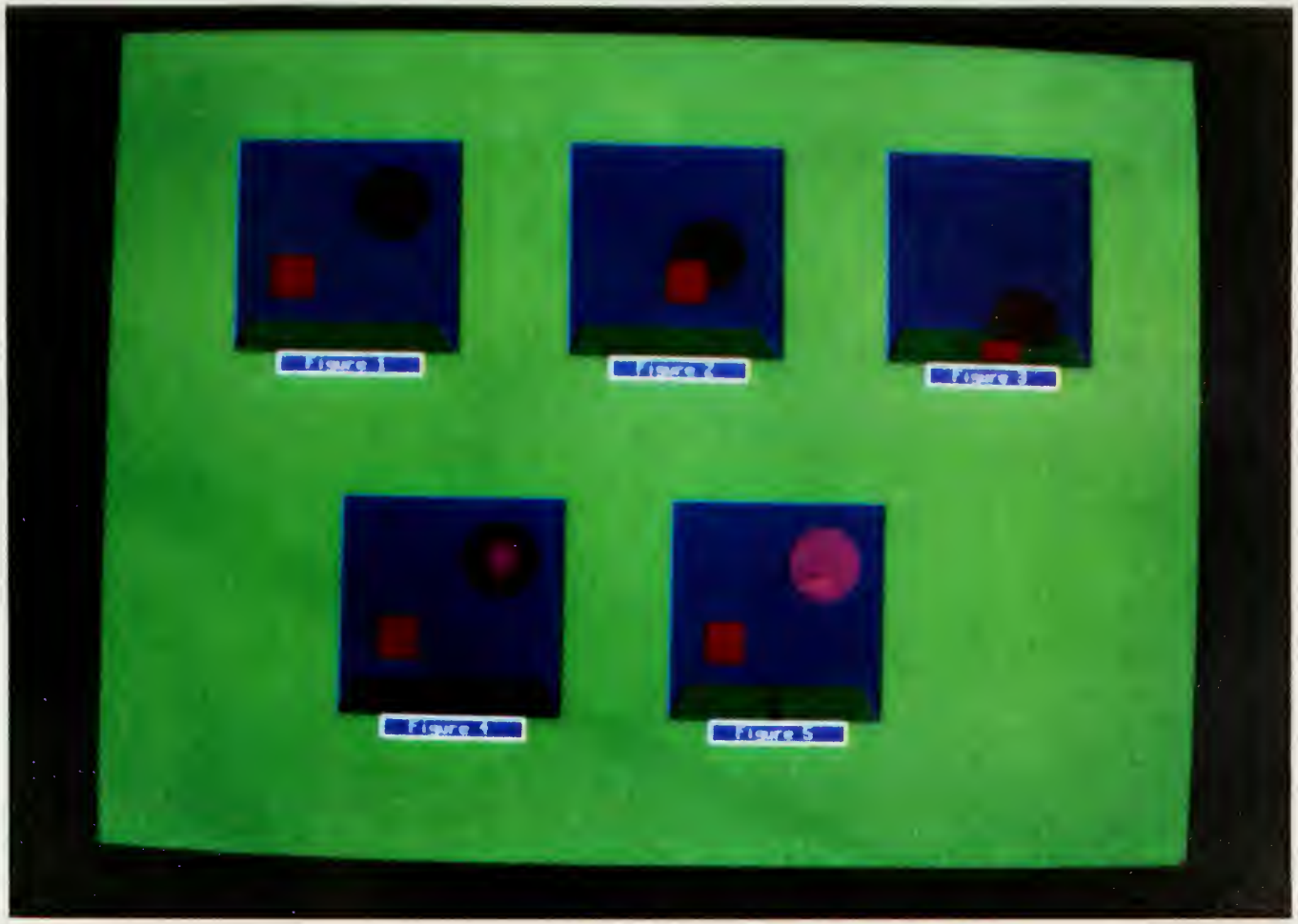

Figure 6.1 - Sample Generated Scenes

then points to the six polygon records that where used to construct the cube. Each one of these records has its own vertex arrays. The second object is a sphere. This object has one subobject, and the subobject has one common part. Since a sphere can be its own bounding volume and since it is not constructed of polygons, then the object chain for the sphere need go no further than the common parts array. The third object is the floor.* This consists of one subobject, one common part and one polygon. All objects in the scene are opaque and have a highly reflective surface.

B. OUTPUT

The output generated by the ray tracer is in the form of a bitmap, with values for each of the red, green, and blue components. These values range from 0 to 1. To display these on the RGB monitor of the IRIS, each red, green, and blue component is then multiplied by 255 and assigned an index in the color table.

*The floor is at a 10 degree angle to the screen to provide a better perspective.

VII. CONCLUSIONS

A. AREAS OF FUTURE RESEARCH

The ray tracer is a powerful tool in computer graphics. In its original design, it produced the finest rendered pictures at that time. Since then there have been numerous extensions, among the most widely known are the ones by Phong, Blinn and Newell, Kay, and Whitted each of which have further enhanced the performance of the ray tracer [Ref. 2: pp. 343-344]. There are two main areas for future research: global illumination models, and intersection algorithms. Both of the areas are of great interest in the graphics world. Since ray tracers and global illumination models can be integrated so easily, working on either problem would undoubtedly lead to insight into the other. This would also be very easy to do because ray tracing naturally lends itself to a modular design making it easy to establish hooks for the testing of a large number of algorithms, both illumination and intersection.

B. CONCLUSIONS

We have examined the three major areas of ray tracing: the scene data needed, the intersection problem, and the intensity problem. The data structure used was adapted from [Ref. 3] and proved to be useful. The intersection problem, although involved, is not complex. The algorithms used in this

implementation are simple. The inclusion of a simple global illumination model, was easy to integrate and provided fair results. The ray tracer provides an excellent test bed program and implemented can provide a useful tool to study numerous problems, not only in lighting and shading but also in intersection determination.

APPENDIX A – SOURCE LISTINGS

DECLARATIONS

```
const
{ THIS IS THE CONSTANTS DECLARATION SECTION }
  maximum_size_of_stack = 100;
  initial_pixel_z = 0;

type
{ TYPES DECLARATION AREA }

  vertices_array = array [1..4] of real;

  polygon_rec = record
    num_vertices   : integer;
    vertice_x,      vertice_y,      vertice_z       : vertices_array;
    surface_normal_x, surface_normal_y, surface_normal_z : real
  end;

  polygon_ptr = ^polygon_array;
  polygon_array = array [1..6] of polygon_rec;

  common_part_rec = record
    K_ar, K_ag, K_ab : real;   { AMBIENT DIFFUSE COEFFICIENT }
    K_dr, K_dg, K_db : real;   { DIRECT DIFFUSE COEFFICIENT }
    K_sr, K_sg, K_sb : real;   { SPECULAR COEFFICIENT }
    K_tr, K_tg, K_tb : real;   { TRANSMISSION COEFFICIENT }
    obj_refraction_index : real; { OBJECTS REFRACTION COEFFICIENT }
    obj_phong_exp    : integer;  { PHONG'S SPECULAR EXPONENT }
    num_polygons : integer;
    polygons : polygon_ptr
  end;

  common_part_ptr = ^common_part_array;
  common_part_array = array [1..3] of common_part_rec;


  sub_object_rec = record
    num_common_parts  : integer:
    common_parts      : common_part_ptr;
    sub_bsphere_radius: real;  { RADIUS OF SUBOBJECTS BOUNDING SPHERE }
    sub_bsphere_x     : real;  { CENTER OF BOUNDING SPHERE }
    sub_bsphere_y     : real;
    sub_bsphere_z     : real;
    subobj_type       : integer;      { 0: SPHERE, 1: PLANAR-POLYGON }
  end;

  sub_object_ptr = ^sub_object_array;
```

```
  sub_object_array = array [1..3] of sub_object_rec;

  object_rec = record
    num_sub_objects   : integer;
    sub_objects       : sub_object_ptr;
    obj_bsphere_radius : real; { RADIUS OF OBJECTS BOUNDING SPHERE }
    obj_bsphere_x     : real; { CENTER OF BOUNDING SPHERE }
    obj_bsphere_y     : real;
    obj_bsphere_z     : real;
    opcode            : integer  { CURRENTLY NOT USED }
  end;

  object_ptr = ^object_array;
  object_array = array [1..4] of object_rec;

  light_rec = record
    I_r, I_g, I_b :  real;              { INTENSITY OF THE LIGHT }
    light_x, light_y, light_z : real;    { LIGHT POSITION }
    dimension1, dimension2 : real        { NOT USED }
  end;

  light_ptr = ^light_array;
  light_array = array [1..3] of light_rec;

  picture_rec = record
    num_objs        : integer;
    objects         : object_ptr;
    num_lights      : integer;
    lights          : light_ptr;
    global_refraction_index      : real;
    no_zero         : real;
    ambient_r       : real;
    ambient_g       : real;
    ambient_b       : real;
    background_color_r : real;
    background_color_g : real;
    background_color_b : real;
    view_position_x    : real;
    view_position_y    : real;
    view_position_z    : real;
    screen_max_x       : integer;
    screen_max_y       : integer;

  end;

{ ************************************************************************** }

  raytype   = (none, view, reflected, refracted);
  colortype = (red, green, blue);

{ RAY DATA RECORD }
  ray_ptr = ^ray_rec;
```

```
  ray_rec = record
    ray_type        : raytype;
    ray_origin_x    : real;    { ORIGIN OF RAY }
    ray_origin_y    : real;
    ray_origin_z    : real;
    ray_vector_x    : real;    { DIRECTION OF RAY }
    ray_vector_y    : real;
    ray_vector_z    : real;
    ray_stype       : raytype; { TYPE OF SOURCE RAY }
    intersection_flag : boolean;
    obj_idx         : integer; { PATH DECSRIBING OBJECT INTERSECTED }
    subobj_idx      : integer;
    cpart_idx       : integer;
    polygon_idx     : integer;
    intersection_x  : real;    { INTERSECTION POINT }
    intersection_y  : real;
    intersection_z  : real;
{ DISTANCE BETWEEN RAY'S ORIGIN AND INTERSECTION POINT }
    d               : real;
{
 INTENSITY OF LIGHT COMING IN ALONG THE REFRACTED RAY GENERATED BY THIS
 RAY
}
    I_tr, I_tg, I_tb  : real;
{
 INTENSITY OF LIGHT COMING IN ALONG THE REFLECTED RAY GENERATED BY THIS
 RAY
}

    I_sr, I_sg, I_sb  : real;
    ray_link        : ray_ptr
  end; { ray_ptr }
{.PA}
{ ********************************** VAR ************************************ }
var

  outfile : text;
  sysin   : text;
  sysout  : text;

{ USED IN CONVERTING ALL RAYS/VECTORS TO UNIT VECTORS }
  x, y, z : real;
  unitx, unity, unitz : real;
  dist : real;

  ray_number : integer;

{ USED AS SHORTHAND BECAUSE THE OBJECT PATHS GET LONG }
  cpart_path : common_part_ptr;
  subobj_path : sub_object_ptr;

  color   : colortype;
```

```
{ USED TO IMPLEMENT THE STACK }
  ray_top,
  ray_current,
  ray_next     : ray_ptr;

  intensity_red,
  intensity_green,
  intensity_blue  : real;

  pixel_x,
  pixel_y,
  pixel_z   : integer;

  ray_generation_number : integer;

  intersection_point_x,
  intersection_point_y,
  intersection_point_z   : real;

  old_intersection_point_x,
  old_intersection_point_y,
  old_intersection_point_z : real;

  temp_integer1,
  temp_integer2,
  temp_integer3   : integer;

  reflected_ray_x,
  reflected_ray_y,
  reflected_ray_z  : real;

  refracted_ray_x,
  refracted_ray_y,
  refracted_ray_z  : real;

  surface_normal_x,
  surface_normal_y,
  surface_normal_z  : real;

{ USED TO INITIALIZE RAYS }
  initial_ray_type   : raytype;
  initial_ray_origin_x, initial_ray_origin_y, initial_ray_origin_z : real;
  initial_ray_vector_x, initial_ray_vector_y, initial_ray_vector_z : real;
  initial_ray_stype     : raytype;
  initial_intersection_flag  : boolean;
  initial_obj_idx          : integer;
  initial_subobj_idx        : integer;
  initial_cpart_idx         : integer;
  initial_polygon_idx       : integer;
  initial_intersection_x,
  initial_intersection_y,
  initial_intersection_z     : real;
```

```
  initial_d    : real;
  initial_I_tr, initial_I_tg, initial_I_tb : real;
  initial_I_sr, initial_I_sg, initial_I_sb : real;

{ HOLDS CURRENT RAY WHILE IN ACTUAL RAY TRACING LOOP }
  current_ray_type   : raytype;
  current_ray_origin_x, current_ray_origin_y, current_ray_origin_z : real;
  current_ray_vector_x, current_ray_vector_y, current_ray_vector_z : real;
  current_ray_stype    : raytype;
  current_intersection_flag  : boolean;
  current_obj_idx          : integer;
  current_subobj_idx        : integer;
  current_cpart_idx         : integer;
  current_polygon_idx       : integer;
  current_intersection_x,
  current_intersection_y,
  current_intersection_z    : real;
  current_d    : real;
  current_I_tr, current_I_tg, current_I_tb : real;
  current_I_sr, current_I_sg, current_I_sb : real;

  reflected_ray : boolean;
  refracted_ray : boolean;

  source_ray_num  : integer;
  source_ray_type : raytype;

{ POINTER TO PICTURE RECORD }
  picture : picture_rec;

{
 USED TO ESTABLISH LINKED LIST OF OBJ, SUBOBJ, LIGHTS, CPARTS, AND
 POLYGONS.
}
  light_cntr : integer;
  light_current : light_ptr;

  obj_cntr : integer;
  obj_curr : object_ptr;

  subobj_cntr : integer;
  subobj_curr  : sub_object_ptr;

  cpart_cntr : integer;
  cpart_curr  : common_part_ptr;

  poly_cntr : integer;
  poly_next,
  poly_curr  : polygon_ptr;

  vertice_cntr : integer;
```

```
{ USED WHEN READING IN DATA }
  poly_loop_cnt : integer;
  light_loop_cnt : integer;
  cpart_loop_cnt : integer;
  subobj_loop_cnt : integer;
  object_loop_cnt : integer;
  vertice_loop_cnt : integer;

  intersection_X, intersection_Y, intersection_Z : real;

{ USED TO IDENTIFY INTERSECTED OBJECTS }
  object_idx,
  subobj_idx,
  cpart_idx,
  polygon_idx : integer;

  intersection_flag : boolean;

{ ********************************************************************************* }
```

```
{
***************************** INTPRCS6.PAS ******************************
*   These are the intersection procedures used in the ray tracer program
*************************************************************************
}
{
************************ SPHERE INTERSECTION ****************************
* CALLED FORM : CHECK_FOR_SUBOBJ-INTERSECTION
* CALLS TO : NONE
* DESC : CALCULATES THE INTERSECTION POINT BETWEEN A RAY/VECTOR AND A
*      SPHERE.
* INPUT : The centerpoint of the sphere and it's radius. The direction of
*       the ray and a known point on the ray -- which would be it's
*       origin.
* OUTPUT : A flag indicating whether or not there was an intersection and
*        if there was the actual intersection point itself.
*************************************************************************
}
procedure sphere_intersection (Px, Py, Pz : real;
                               Vx, Vy, Vz : real;
                               Cx, Cy, Cz : real;
                               r          : real;
                           var o_intersection_flag : boolean;
                           var Sphere_x, Sphere_y, Sphere_z : real );

var
  a, b, c, t1, t2      : real;
  X1, Y1, Z1           : real;
  X2, Y2, Z2           : real;
  distance1, distance2 : real;
  radical : real;
  diffx, diffy, diffz : real;

begin

{ INITIALIZE }
  Sphere_x := 0;
  Sphere_y := 0;
  Sphere_z := 0;
  o_intersection_flag := false;

{ SET UP THE COMPONENTS OF THE QUADRATIC EQUATION }
  a := ( sqr(Vx) + sqr(Vy) + sqr(Vz));
  b := ( (2 * Px * Vx) +
       (2 * Py * Vy) +
       (2 * Pz * Vz) -
       (2 * Cx * Vx) -
       (2 * Cy * Vy) -
       (2 * Cz * Vz) );
```

```
  c := ( sqr(Cx) + sqr(Cy) + sqr(Cz) +
       sqr(Px) + sqr(Py) + sqr(Pz) -
       (2 * Cx * Px) - (2 * Cy * Py) - (2 * Cz * Pz) - sqr(r));

    radical := (sqrt ( sqr(b) - (4 * a * c)));

{ START COMPUTATIONS ON QUADRATIC EQUATION }
     if (radical < 0) then  begin
       writeln('WARNING - imagionary number possible in SPHERE INTERSECTION');
       o_intersection_flag := false
     end
    else  begin
       if (radical = 0) then begin
{
IF 0 THEN JUST ONE INTERSECTION POINT(THE LINE IS TANGENT TO THE SPHERE)
}
         t1 := (-b / 2 * a);          { SOLVE FOR t }
         Sphere_x := Px + (Vx * t1);  { CALCULATE POINT USING t }
         Sphere_y := Py + (Vy * t1);
         Sphere_z := Pz + (Vz * t1);
         o_intersection_flag := true
       end
       else  begin
{
 THERE WERE TWO INTERSECTION POINTS -- ONE ENTERANCE POINT AND ONE EXIT
POINT. SOLVE FOR BOTH ts'
}
         t1 :=  (-b + (sqrt ( sqr(b) - (4 * a * c)))) / 2 * a;
         t2 :=  (-b - (sqrt ( sqr(b) - (4 * a * c)))) / 2 * a;

         X1 := Px + (Vx * t1);
         Y1 := Py + (Vy * t1);
         Z1 := Pz + (Vz * t1);

{
CALCULATE DISTANCE FOR BOTH INTERSECTION POINTS FROM THE POINT OF ORIGIN
}
distance1 := (sqrt (sqr(X1 - Px) + sqr(Y1 - Py) + sqr(Z1 - Pz)));

         X2 := Px + (Vx * t2);
         Y2 := Py + (Vy * t2);
         Z2 := Pz + (Vz * t2);

         distance2 := (sqrt (sqr(X2 - Px) + sqr(Y2 - Py) + sqr(Z2 - Pz)));

{ COMPARE DISTANCES AND SELECT THE INTERSECTION POINT THAT IS CLOSER }
         if distance1 < distance2 then  begin
           Sphere_x := X1;
           Sphere_y := Y1;
           Sphere_z := Z1;
         end
         else  begin
```

```
            Sphere_x := X2;
            Sphere_y := Y2;
            Sphere_z := Z2;
          end;

          o_intersection_flag := true
        end; { ELSE }

        diffx := Px - Sphere_x;
        diffy := Py - Sphere_y;
        diffz := Pz - Sphere_z;

{
 A CHECK TO INSURE THAT THE POINT SELECTED ISN'T THE RAYS POINT OF ORIGIN
}
        if ((diffx <= 0.0000) and (diffy <= 0.0000) and (diffz <= 0.0000)) then
          o_intersection_flag := false;

      end;{ ELSE }
end; {* SPHERE INTERSECTIONS *}


{.PA}
{
 ********************** CALCULATE PLANE EQUATION ***************************
 * CALLED FROM : FIND_INTERSECTED_POLYGON
 * CALLS TO : NONE
 * DESC : Calculates the constants of the equation of a plane when given
 *      three point on the plane.
 * INPUT : Three vertices of a planar polygon.
 * OUTPUT : The A, B, C, and D constants for the equation of a plane.
 *******************************************************************************
}

procedure calculate_plane_equation(X1, Y1, Z1  : real;
                                   X2, Y2, Z2  : real;
                                   X3, Y3, Z3  : real;
                                  var A,B,C,D : real );

begin

  A := Y1 * (Z2 - Z3) + Y2 * (Z3 - Z1) + Y3 * (Z1 - Z2);
  B := Z1 * (X2 - X3) + Z2 * (X3 - X1) + Z3 * (X1 - X2);
  C := X1 * (Y2 - Y3) + X2 * (Y3 - Y1) + X3 * (Y1 - Y2);
  D := -X1 * ((Y2 * Z3) - (Y3 * Z2)) - (X2 * ((Y3 * Z1) - (Y1 * Z3)))
   - (X3 * ((Y1 * Z2) - (Y2 * Z1)));


end; { calculate_plane_equation }


{
```

```
******************** FIND INTERSECTION POINT ************************
* CALLED FROM: FIND_INTERSECTED_POLYGON,
*          CHECK_FOR_SUBOBJ_INTERSECTION
*          CHECK_FOR_INTERSECTION
* CALLS TO : NONE
* DESC : Calculates the intersection point between a ray/vector and
*      a plane.
* INPUT : Ray direction and a known point on the ray, i.e., it's
*       point of origin. The constants (A,B,C,D) of the equation
*       of a plane.
* OUTPUT : The intersection point.
*************************************************************************
}
procedure find_intersection_point(i_A, i_B, i_C, i_D       : real;
                          i_ray_x, i_ray_y, i_ray_z  : real;
                          i_source_x, i_source_y, i_source_z : real;
                        var o_intersection_point_x,
                          o_intersection_point_y,
                          o_intersection_point_z     : real);

var
  t  : real;

begin

{ SET UP FOR FINDING t FROM THE EQUATION FOR A LINE AND A PLANE }
  t := (i_D - ((i_A * i_source_x) + (i_B * i_source_y) + (i_C * i_source_z))) /
         ((i_A * i_ray_x) +
          (i_B * i_ray_y) +
          (i_C * i_ray_z));
{
 SUBSTITUTE t BACK INTO THE EQUATION FOR A LINE TO GET THE INTERSECTION
 POINT.
}

  o_intersection_point_x := (i_ray_x * t) + i_source_x;
  o_intersection_point_y := (i_ray_y * t) + i_source_y;
  o_intersection_point_z := (i_ray_z * t) + i_source_z;


end; { find_intersection_point }
{.PA}
{
 ****************************** CALEQ ***********************************
 * CALLED FROM: FIND_INTERSECTED_POLYGON
 *          CHECK_FOR_SUBOBJ_INTERSECTION
 *          CHECK_FOR_INTERSECTION
 * CALLS TO : NONE
 * DESC :  This calculates the constants A,B,C,D of the equation of a plane
 *       given only a point on the plane and the surface normal of the
 *       plane.
 * INPUT : A point and the surface normal of the plane whose equation you
```

```
*       are trying to figure out.
* OUTPUT: The constants A,B,C,D of the equation of a plane.
*************************************************************************
}
procedure  caleq(bsphere_x, bsphere_y, bsphere_z,
            vector_x, vector_y, vector_z : real;
          var  A, B, C, D : real);

begin

  A := vector_x;
  B := vector_y;
  C := vector_z;
  D := (vector_x * bsphere_x) +
     (vector_y * bsphere_y) +
     (vector_z * bsphere_z);

end; { caleq }


{
************************* POLYGON ORIENTATION ****************************
* CALLED FORM: FIND_INTERSECTED_POLYGON
* CALLS TO : NONE
* DESC : This checks to see if the incoming ray will hit the front face of
*       this polygon. It does this by comparing the angle between the
*       surface normal of the polygon and the inverse of the incoming ray.
*       If the angle is greater than 90 degrees then the ray is
*       approaching the back of the polygon.
* INPUT: The direction of the incoming ray. The surface normal of hte
*       object.
* OUTPUT: A boolean value TRUE/FALSE depending on whether or not the
*        polygon is facing the right direction.
*************************************************************************
}

procedure polygon_orientation (view_vector_x, view_vector_y, view_vector_z : real;
                      surface_normal_x, surface_normal_y, surface_normal_z : real;
                    var o_good_orientation : boolean);

var
  cosine_theta        : real;
  length_view_vector   : real;
  length_surface_normal : real;
  dot_product         : real;

begin

{
 TAKE THE DOT PRODUCT OF THE INVERSE OF THE VIEW VECTOR AND THE SURFACE
 NORMAL OF THE POLYGON IN QUESTION.
}
```

```
  dot_product := (- view_vector_x) * surface_normal_x  +
                 (- view_vector_y) * surface_normal_y  +
                 (- view_vector_z) * surface_normal_z;

{ CALCULATE THE MAGNITUDE OF THE VECTORS }
  length_view_vector := sqrt(sqr(- view_vector_x) +
                             sqr(- view_vector_y) +
                             sqr(- view_vector_z));

  length_surface_normal := sqrt(sqr( surface_normal_x) +
                                sqr( surface_normal_y) +
                                sqr( surface_normal_z));

{ CALCULATE THE COSINE OF THE ANGLE BETWEEN THE RAYS }
  cosine_theta := dot_product / (length_view_vector * length_surface_normal);

  if (cosine_theta > 0) then
    o_good_orientation := true
  else
    o_good_orientation := false;

end; { polygon_orientation }


{.PA}
{
 ************************ FIND INTERSECTED POLYGON ************************
 * CALLED FROM : CHECK_FOR_SUBOBJ_INTERSECTION
 * CALLS TO : POLYGON_ORIENTATION
 *         CALEQ
 *         FIND_INTERSECTION_POINT
 *         CALCULATE_PLANE_EQUATION
 * DESC : This determines if there is an intersection between a line/ray
 *      and a polygon. If there is it calculates what it is.
 * INPUT: The direction of the shooting ray, a known point on that ray -
 *      it's origin, and the object path identifying the subobject to
 *      examine.
 * OUTPUT: A flag indicating whether or not a polygon was hit. If one was
 *       hit the path identifying which one it was and the actual
 *       intersection point itself.
 ***************************************************************************
}
procedure find_intersected_polygon(i_ray_x, i_ray_y, i_ray_z : real;
                          i_source_x, i_source_y, i_source_z : real;
                          i_obj_idx,
                          i_subobj_idx : integer;
                        var o_polygon_intersection_x,
                          o_polygon_intersection_y,
                          o_polygon_intersection_z : real;
                        var o_cpart_idx,
                          o_polygon_idx : integer;
                        var o_intersection_flag : boolean);
```

```
type
{
  THIS IS SET UP TO HANDLE 6 SIDED POLYGONS. AS EACH SIDE OF THE POLYGON
 IS TESTED TO SEE WHETHER THE INTERSECTION POINT LIES INSIDE OR OUTSIDE OF
 IT THE CORRESPONDING ELEMENT IN THE ARRAY IS SET EITHER TRUE OR FALSE. AN
 ARRAY OF ALL TRUE MEANS THAT THE INTERSECTION POINT LIES WITHIN THE
 POLYGON.
}
  intersection_array = array[1..6] of boolean;

var
  point_outside_polygon : boolean;
  intersections : intersection_array;

  cpart_cnt,
  polygon_cnt,
  vertice_cnt : integer;

  intersection_found,
  good_orientation : boolean;

  cpart_path :  common_part_ptr;

  polygon_path :  polygon_ptr;

{ USED FOR ARBITRARY POINT TO DEFINE BOUNDING PLANE }
  anchor_x, anchor_y, anchor_z : real;

  markerD,
  check_point_D : real;

  polygonX, polygonY, polygonZ : real;

  A, B, C, D : real;
  x1, y1, z1 : real;
  x2, y2, z2 : real;
  x3, y3, z3 : real;

  diffx, diffy, diffz : real;

  loop_cnt : integer;

begin

{ INITIALIZE }
  o_cpart_idx := 0;
  o_polygon_idx := 0;
  o_polygon_intersection_x := 0.0;
  o_polygon_intersection_y := 0.0;
  o_polygon_intersection_z := 0.0;
  o_intersection_flag := false;
```

```
  cpart_cnt := 1;
  polygon_cnt := 1;
  vertice_cnt := 1;
  intersection_found := false;
  good_orientation := false;
  cpart_path := picture.objects^[i_obj_idx].sub_objects^[i_subobj_idx].
            common_parts;
  polygon_path := cpart_path^[cpart_cnt].polygons;

{
 THIS CHECKS EACH COMMON PART FOR AN INTERSECTION WITH ONE OF IT'S
POLYGONS.
}
  repeat

{ THIS LOOP CHECKS EACH POLYGON OF A COMMON PART FOR INTERSECTION }
    repeat
      point_outside_polygon := false;
      polygon_orientation(i_ray_x, i_ray_y, i_ray_z,
                  polygon_path^[polygon_cnt].surface_normal_x,
                  polygon_path^[polygon_cnt].surface_normal_y,
                  polygon_path^[polygon_cnt].surface_normal_z,
                  good_orientation);

      if good_orientation then  begin

        caleq(polygon_path^[polygon_cnt].vertice_x[1],
            polygon_path^[polygon_cnt].vertice_y[1],
            polygon_path^[polygon_cnt].vertice_z[1],
            polygon_path^[polygon_cnt].surface_normal_x,
            polygon_path^[polygon_cnt].surface_normal_y,
            polygon_path^[polygon_cnt].surface_normal_z,
            A,B,C,D);


        find_intersection_point(A, B, C, D,
                        i_ray_x,
                        i_ray_y,
                        i_ray_z,
                        i_source_x, i_source_y, i_source_z,
                        polygonX, polygonY, polygonZ);

{
 CHECK TO MAKE SURE YOU ARE NOT CONSIDERING THE SOURCE POINT(ORIGIN) OF
THE RAY.
}
        diffx := i_source_x - polygonX;
        diffy := i_source_y - polygonY;
        diffz := i_source_z - polygonZ;
{
 THIS SETS A FLAG IF YOU DO CONSIDER THE SAME POINT, THE CHECKS IN THE
OTHER PROCEDURE SHOULD PREVENT THIS BUT JUST IN CASE.
```

```
}
        if ((diffx <= 0.000) and (diffy <= 0.000) and (diffz <= 0.000))
         then  begin

          { writeln(sysout,'set trip-wire');}
           intersections[1] := false
         end
        else  begin
{
  THIS LOOP CHECKS EACH EDGE OF THE POLYGON TO SEE IF THE INTERSECTION
 POINT LIES INSIDE OR OUTSIDE OF IT.
}
           repeat
{ THIS SELECTS THE FIRST VERTEX OF A POLYGON }
             x1 := polygon_path^[polygon_cnt].vertice_x[vertice_cnt];
             y1 := polygon_path^[polygon_cnt].vertice_y[vertice_cnt];
             z1 := polygon_path^[polygon_cnt].vertice_z[vertice_cnt];

             if (vertice_cnt = (polygon_path^[polygon_cnt].num_vertices -
                          1))
{
 WHEN YOU  PICK THE NEXT TO LAST VERTEX YOU CAN SELECT THE NEXT
 CONSECUTIVE VERTEX TO ESTABLISH THE EDGE THROUGH WHICH YOU WANT THE
 BOUNDING PLANE TO PASS. YOU THEN MUST PICK ONE OF THE OTHER VERTICES (AND
 IT DOES NOT MAKE ANY DIFFERENCE WHICH ONE. I PICK THE FIRST ONE.) TO BE
 USED TO PUT INTO THE EQUATION OF THE PLANE THE RESULT OF WHICH IS COMPARED
 AGAINST THE RESULT THAT COMES FROM PLUGGING THE INTERSECTION POINT INTO
 THE EQUATION OF THE BOUNDING PLANE.
}
              then  begin
               x2 := polygon_path^[polygon_cnt].
                     vertice_x[vertice_cnt + 1];
               y2 := polygon_path^[polygon_cnt].
                     vertice_y[vertice_cnt + 1];
               z2 := polygon_path^[polygon_cnt].
                     vertice_z[vertice_cnt + 1];

               x3 := polygon_path^[polygon_cnt].vertice_x[1];
               y3 := polygon_path^[polygon_cnt].vertice_y[1];
               z3 := polygon_path^[polygon_cnt].vertice_z[1]
              end
             else  begin
{
 IF YOU DO NOT HAVE THE NEXT TO LAST EDGE THEN JUST SELECT THE NEXT
 CONSECUTIVE VERTICE TO ESTABLISH THE EDGE FOR THE BOUNDING PLANE AND
 THE ONE AFTER THAT TO PLUG INTO THE EQUATON OF THE PLANE.
}
               x2 := polygon_path^[polygon_cnt].
                     vertice_x[vertice_cnt + 1];
               y2 := polygon_path^[polygon_cnt].
                     vertice_y[vertice_cnt + 1];
               z2 := polygon_path^[polygon_cnt].
```

```
                vertice_z[vertice_cnt + 1];

            x3 := polygon_path^[polygon_cnt].
                vertice_x[vertice_cnt + 2];
            y3 := polygon_path^[polygon_cnt].
                vertice_y[vertice_cnt + 2];
            z3 := polygon_path^[polygon_cnt].
                vertice_z[vertice_cnt + 2]
        end;

        if (vertice_cnt = polygon_path^[polygon_cnt].num_vertices )
{
 IF THE VERTICE SELECTED IS THE LAST ONE THEN JUST PICK THE FIRST VERTICE
TO ESTABLISH YOUR BOUNDING EDGE AND THE SECOND VERTICE TO PLUG INTO THE
EQUATION OF THE PLANE.
}
         then  begin
            x2 := polygon_path^[polygon_cnt].vertice_x[1];
            y2 := polygon_path^[polygon_cnt].vertice_y[1];
            z2 := polygon_path^[polygon_cnt].vertice_z[1];

            x3 := polygon_path^[polygon_cnt].vertice_x[2];
            y3 := polygon_path^[polygon_cnt].vertice_y[2];
            z3 := polygon_path^[polygon_cnt].vertice_z[2]
        end;

{
 ESTABLISH THE ARBITRARY POINT THROUGH WHICH THE PLANE WILL PASS.
}
            anchor_x := ((x2 + x1) / 2) + 10;
            anchor_y := ((y2 + y1) / 2) + 10;
            anchor_z := ((z2 + z1) / 2) + 10;

        calculate_plane_equation( x1, y1, z1,
                                  x2, y2, z2,
                                  anchor_x, anchor_y, anchor_z,
                                  A, B, C, D);

{
 THE RESULT OF PLUGGING IN THE VERTICE OF THE POLYGON INTO THE EQUATION
OF THE PLANE.
}
        markerD := (A * x3) + (B * y3) + (C * z3);
{
 THE RESULT OF PLUGGING IN THE INTERSECTION POINT INTO THE EQUATION OF
THE PLANE.
}
        check_point_D := (A * polygonX) +
                         (B * polygonY) +
                         (C * polygonZ);

        if ((markerD <= -D) and (check_point_D <= -D)) then
```

```
{
 IF THE RESULTS HAVE THE SAME SIGN THEN THEY BOTH LIE ON THE SAME SIDE OF
 THE BOUNDING PLANE. HENCE THE INTERSECTION POINT LIES WITHIN THE POLYGON
 WITH RESPECT TO THAT EDGE.
}
              intersections[vertice_cnt] := true
            else
{
 IF THE RESULTS DON'T HAVE THE SAME SIGN THEN THEY LIE ON OPPOSITE SIDES
 OF THE BOUNDING PLANE. AT THIS POINT THE INTERSECTION POINT HAS BEEN
 PROVEN TO LIE OUTSIDE THE POLYGON.
}
              if ((markerD >= -D) and (check_point_D >= -D)) then
                intersections[vertice_cnt] := true
              else
                intersections[vertice_cnt] := false;
            vertice_cnt := vertice_cnt + 1

          until (vertice_cnt > polygon_path^[polygon_cnt].num_vertices);
        end
      end;

{
  CHECK THE POLYGON INTERSECTION ARRAY TO SEE IF THE INTERSECTION POINT
 FAILED THE INSIDE TEST FOR ANY OF THE EDGES.
}
      for loop_cnt := 1 to polygon_path^[polygon_cnt].num_vertices do
        if not(intersections[loop_cnt]) then
          point_outside_polygon := true;

      if point_outside_polygon then
        intersection_found := false
      else
        intersection_found := true;

      polygon_cnt := polygon_cnt + 1;
    until ((polygon_cnt > cpart_path^[cpart_cnt].num_polygons) or
         (intersection_found));

    cpart_cnt := cpart_cnt + 1;
  until ((cpart_cnt > picture.objects^[i_obj_idx].sub_objects^[i_subobj_idx].
       num_common_parts) or (intersection_found));

{ SET UP THE OUTPUT FOR THE PROCEDURE }
  o_cpart_idx := (cpart_cnt - 1);
  o_polygon_idx := (polygon_cnt - 1);
  o_polygon_intersection_x := polygonX;
  o_polygon_intersection_y := polygonY;
  o_polygon_intersection_z := polygonZ;
  o_intersection_flag := intersection_found;
```

```
end; { find_intersected_polygon }


{.PA}
{
 ******************** CHECK FOR SUBOBJ INTERSECTION ************************
 * CALLED FROM: CHECK_FOR_INTERSETION
 * CALLS TO : CALEQ
 *         FIND_INTERSECTION_POINT
 *         SPHERE_INTERSECTION_POINT
 *         FIND_INTERSECTION_POINT
 * DESC :   Check to find out if the shooting ray intersects this
 *       subobject's bounding volume.
 * INPUT : The object whose bounding volume has been hit.
 *        The direction of the shooting ray.
 *        The origin of the shooting ray.
 * OUTPUT : A flag indicating whether or not there has been an intersection.
 *         The intersection point -- if there is one.
 *         The path to the intersected object.
 ***************************************************************************
}

procedure check_for_subobj_intersection (i_object_idx     : integer;
                                         i_ray_x, i_ray_y, i_ray_z  : real;
                                         i_source_x,
                                         i_source_y,
                                         i_source_z : real;
                                        var  o_intersection_x,
                                         o_intersection_y,
                                         o_intersection_z : real;
                                        var  o_subobj_idx,
                                         o_cpart_idx,
                                         o_polygon_idx : integer;
                                        var  o_intersection_flag : boolean);

var
  closest_object : real;

{ VECTOR BETWEEN THE RAYS ORIGIN AND INTERSECTION POINT }
  view_polygon_vector_x,
  view_polygon_vector_y,
  view_polygon_vector_z : real;

  distance_from_intersection,
  distance_from_view_position      : real;

  subobj_path : sub_object_ptr;
  obj_path : object_ptr;

  cpart_cnt,
  polygon_cnt,
  subobj_cnt : integer;       { USED TO GO INTO RESPECTIVE ARRAYS }
```

```
  A, B, C, D : real;          { CONSTANTS FOR EQUATION OF A PLANE }

{
 INTERSECTION POINT BETWEEN THE RAY AND PLANE THE BOUNDING CIRCLE IS
 INSCRIBED ON.
}
  bplane_intersection_x,
  bplane_intersection_y,
  bplane_intersection_z : real;

{
  INTERSECTION POINT BETWEEN THE RAY AND A POLYGON.
}
  polygonX, polygonY, polygonZ : real;

  intersection_flag : boolean;


begin

  o_subobj_idx := 0;
  o_cpart_idx  := 0;
  o_polygon_idx   := 0;
  o_intersection_flag := false;
  o_intersection_x := 0.0;
  o_intersection_y := 0.0;
  o_intersection_z := 0.0;

{
 ESTABLISH A DEFAULT DISTANCE WITH WHICH THE ACTUAL DISTANCES WIL BE
 COMPARED.
}
  closest_object := 10000.0;
  subobj_path := picture.objects^[i_object_idx].sub_objects;
  obj_path := picture.objects;

{ THIS LOOP CHECKS EACH SUBOBJECT OF AN OBJECT }
  for subobj_cnt := 1 to obj_path^[i_object_idx].num_sub_objects do  begin


{ FIRST ESTABLISH THE PLANE ON WHICH TO DRAW THE BOUNDING CIRCLE }
    caleq(subobj_path^[subobj_cnt].sub_bsphere_x,
        subobj_path^[subobj_cnt].sub_bsphere_y,
        subobj_path^[subobj_cnt].sub_bsphere_z,
        i_ray_x, i_ray_y, i_ray_z,
        A, B, C, D);

{ FIND THE INTERSECTION POINT ON THAT PLANE }
    find_intersection_point(A,B,C,D,
                    i_ray_x, i_ray_y, i_ray_z,
                    i_source_x, i_source_y, i_source_z,
                    bplane_intersection_x,
```

```
                    bplane_intersection_y,
                    bplane_intersection_z);

{
 DETERMINE THE DISTANCE BETWEEN THE INTERSECTION POINT AND THE CENTER
 POINT OF THE BOUNDING SPHERE/CIRCLE.
}
    distance_from_intersection :=
     sqrt(sqr(subobj_path^[subobj_cnt].sub_bsphere_x -
           bplane_intersection_x) +
         sqr(subobj_path^[subobj_cnt].sub_bsphere_y -
           bplane_intersection_y) +
         sqr(subobj_path^[subobj_cnt].sub_bsphere_z -
           bplane_intersection_z) );

{ IF THE DISTANCE IS LESS THAN OR EQUAL TO THEN YOU HAVE AN INTERSECTION }
    if (distance_from_intersection <=
     subobj_path^[subobj_cnt].sub_bsphere_radius) then  begin

      if (subobj_path^[subobj_cnt].subobj_type = 0) then  begin
{
 IF SUBOBJECT TYPE IS A SPHERE THEN YOU NEED TO USE THIS INTERSECTION
 PROCEDURE.
}
       sphere_intersection(i_source_x, i_source_y, i_source_z,
                    i_ray_x, i_ray_y, i_ray_z,
                    subobj_path^[subobj_cnt].sub_bsphere_x,
                    subobj_path^[subobj_cnt].sub_bsphere_y,
                    subobj_path^[subobj_cnt].sub_bsphere_z,
                    subobj_path^[subobj_cnt].sub_bsphere_radius,
                    intersection_flag,
                    polygonX, polygonY, polygonZ);
        cpart_cnt := 1;
        polygon_cnt := 0;

      end
      else  begin
{ IF IT ISN'T A SPHERE THEN USE THESE }
        find_intersected_polygon(i_ray_x, i_ray_y, i_ray_z,
                        i_source_x, i_source_y, i_source_z,
                        i_object_idx,
                        subobj_cnt,
                        polygonX, polygonY, polygonZ,
                        cpart_cnt,
                        polygon_cnt,
                        intersection_flag);
      end; { * ELSE * }

      if intersection_flag then  begin
{
 IF THERE HAS BEEN AN INTERSECTION THE ESTABLISH THE VECTOR BETWEEN THE
 ORIGIN OF THE RAY AND THE INTERSECTION POINT.
```

```
}
        view_polygon_vector_x := i_source_x - polygonX;
        view_polygon_vector_y := i_source_y - polygonY;
        view_polygon_vector_z := i_source_z - polygonZ;

{
 CALCULATE THE MAGNITUDE (DISTANCE) OF THE RAY.
}
        distance_from_view_position := sqrt(sqr(view_polygon_vector_x) +
                                    sqr(view_polygon_vector_y) +
                                    sqr(view_polygon_vector_z));

        if distance_from_view_position < closest_object then  begin
{
 COMPARE IT AGAINST THE OTHER DISTANCES AND SELECT THE CLOSEST ONE
}
          closest_object := distance_from_view_position;
          o_subobj_idx := subobj_cnt;
          o_cpart_idx  := cpart_cnt;
          o_polygon_idx   := polygon_cnt;
          o_intersection_flag := intersection_flag;
          o_intersection_x := polygonX;
          o_intersection_y := polygonY;
          o_intersection_z := polygonZ
        end
      end
    end
  end; {* FOR *}

end; { check_for_object_intersection }


{.PA}
{
 ************************ CHECK FOR INTERSECTION ****************************
 * CALLED FROM: MAIN AND CALCULATE INTENSITY
 * CALLS TO : CALEQ
 *         FIND_INTERSECTION_POINT
 *         CHECK_FOR_SUBOBJECT_INTERSECTION
 * DESC :  DETERMINES IF THERE IS AN INTERSECTION BETWEEN THE SHOOTING RAY
 *      AND THE BOUNDING VOLUME OF AN OBJECT.
 * INPUT : Direction of the shooting ray.
 *         Origin of the shooting ray.
 *         Pointer into the object array.
 * OUTPUT : Flag indicating whether or not there was an intersection.
 *         If there was an intersection then then the intersection point.
 *         The path to the intersected polygon.
 ****************************************************************************
}

 procedure check_for_intersection (i_ray_x, i_ray_y, i_ray_z : real;
                         i_source_x, i_source_y, i_source_z : real;
```

```
                    i_picture_objects : object_ptr;
                var o_intersection_x,
                    o_intersection_y,
                    o_intersection_z : real;
                var o_object_idx,
                    o_subobj_idx,
                    o_cpart_idx,
                    o_poly_idx    : integer;
                var o_intersection_flag : boolean);

var
  closest_object : real;

  object_cnt,
  object_idx,
  subobj_cnt,
  cpart_cnt,
  polygon_cnt : integer;

  distance_from_viewposition,
  distance_from_intersection  : real;

{
 INTERSECTION POINT BETWEEN SHOOTING RAY AND THE PLANE THE BOUNDING CIRCLE
 IS INSCRIBED ON.
}
  bplane_intersection_x,
  bplane_intersection_y,
  bplane_intersection_z : real;

  view_bplane_vector_x,
  view_bplane_vector_y,
  view_bplane_vector_z : real;

{
 INTERSECTION POINT BETWEEN THE SHOOTING RAY AND A POLYGON.
}
  polygonX, polygonY, polygonZ : real;

{
 CONSTANTS FOR THE EQUATION OF A PLANE.
}
  A, B, C, D : real;

  found_intersection : boolean;

  distance : real;

begin

  o_object_idx := 0;
  o_subobj_idx := 0;
```

```
  o_cpart_idx  := 0;
  o_poly_idx   := 0;
  o_intersection_x := 0.0;
  o_intersection_y := 0.0;
  o_intersection_z := 0.0;
  o_intersection_flag := false;
  closest_object := 10000.0;

  found_intersection := false;

  object_cnt := 1;

{
 LOOP TO CHECK EACH OBJECT IN THE OBJECT ARRAY.
}
  repeat

    if (((picture.objects^[object_cnt].obj_bsphere_z > i_source_z) and
        (i_ray_z > 0 )) or
       ((picture.objects^[object_cnt].obj_bsphere_z < i_source_z) and
        (i_ray_z < 0 ))) then begin

{
 ESTABLISH PLANE ON WHICH TO DRAW BOUNDING CIRCLE.
}
      caleq(picture.objects^[object_cnt].obj_bsphere_x,
          picture.objects^[object_cnt].obj_bsphere_y,
          picture.objects^[object_cnt].obj_bsphere_z,
          i_ray_x, i_ray_y, i_ray_z,
          A, B, C, D);

{
 FIND INTERSECTION POINT BETWEEN THAT PLANE AND THE SHOOTING RAY.
}
      find_intersection_point(A,B,C,D,
                      i_ray_x, i_ray_y, i_ray_z,
                      i_source_x, i_source_y, i_source_z,
                      bplane_intersection_x,
                      bplane_intersection_y,
                      bplane_intersection_z);

{
 DETERMINE DISTANCE BETWEEN THE CENTER OF THE CIRCLE AND THE
 INTERSECTION POINT.
}
      distance_from_intersection :=
       sqrt(sqr(picture.objects^[object_cnt].obj_bsphere_x -
            bplane_intersection_x) +
          sqr(picture.objects^[object_cnt].obj_bsphere_y -
            bplane_intersection_y) +
          sqr(picture.objects^[object_cnt].obj_bsphere_z -
            bplane_intersection_z) );
```

```
{
 IF INTERSECTION POINT LIES WITHIN CIRCLE THEN START CHECKING THE
SUBOBJECTS THAT MAKE UP THE OBJECT.
}
      if distance_from_intersection <=
       picture.objects^[object_cnt].obj_bsphere_radius then  begin

        check_for_subobj_intersection (object_cnt,
                                i_ray_x, i_ray_y, i_ray_z,
                                i_source_x, i_source_y, i_source_z,
                                polygonX, polygonY, polygonZ,
                                subobj_cnt,
                                cpart_cnt,
                                polygon_cnt,
                                found_intersection);

{
 DETERMINE THE DISTANCE BETWEEN THE ORIGIN AND INTERSECTION POINT OF
 OF THE RAY
}
        if found_intersection then  begin
          view_bplane_vector_x := i_source_x - bplane_intersection_x;
          view_bplane_vector_y := i_source_y - bplane_intersection_y;
          view_bplane_vector_z := i_source_z - bplane_intersection_z;

          distance_from_viewposition := sqrt(sqr(view_bplane_vector_x) +
                                    sqr(view_bplane_vector_y) +
                                    sqr(view_bplane_vector_z));

{
  SELECT ONE CLOSEST TO RAY'S ORIGIN.
}
          if distance_from_viewposition < closest_object then  begin
            closest_object := distance_from_viewposition;
            o_object_idx := object_cnt;
            o_subobj_idx := subobj_cnt;
            o_cpart_idx  := cpart_cnt;
            o_poly_idx   := polygon_cnt;
            o_intersection_x := polygonX;
            o_intersection_y := polygonY;
            o_intersection_z := polygonZ;
            o_intersection_flag := found_intersection
          end
        end
      end;

    end;
    object_cnt := object_cnt + 1;
  until  (object_cnt > picture.num_objs);

end; { check_for_intersection }
```

```
{
***************************** STACK_EMPTY ****************************
* CALLED FROM : MAIN and POP
* CALLS TO    : NONE
* DESC    : CHECKS TO SEE IF STACK EMPTY
* INPUT   : POINTER TO TOP OF STACK
* OUTPUT  : BOOLEAN VALUE - TRUE/FALSE
**********************************************************************
}
function stack_empty (input_ray_top : ray_ptr) :boolean;
begin
  STACK_EMPTY := input_ray_top = nil;

end; { STACK_EMPTY }
```

```
{
************************** STACK EXCEEDED ***************************
* CALLED FROM : MAIN
* CALLS TO    : NONE
* DESC    : CHECKS TO SEE IF STACK IS FULL
* INPUT   : CURRENT SIZE OF STACK and MAXIMUM SIZE OF STACK
* OUTPUT  : BOOLEAN VALUE  TRUE/FALSE
**********************************************************************
} function stack_exceeded(input_t1,
                  input_t2 : integer) : boolean;

begin
  if input_t1 > input_t2 then
    stack_exceeded := true
  else
    stack_exceeded := false;


end; { STACK EXCEEDED }
```

```
{.PA}
{
************************ CALCULATE_REFRACTED_RAY *************************
* CALLED FROM : MAIN
* CALLS TO    : NONE
* DESC    : This calculates the direction of a refracted ray.
```

```
* INPUT   : Direction of source ray, surface normal of intersected object,
*          the objects index of refraction, the global index of refraction.
* OUTPUT  : A flag set to true or false, depending on weither or not a
*          refracted ray was created. If one was then it's direction is
*          given.
*******************************************************************************
}
procedure calculate_refracted_ray (i_ray_vector_x,
                                   i_ray_vector_y,
                                   i_ray_vector_z        : real;
                                   i_obj_surface_normal_x,
                                   i_obj_surface_normal_y,
                                   i_obj_surface_normal_z : real;
                                   i_obj_ridx      : real;
                                   i_global_ridx   : real;
                                 var o_refracted_ray_x,
                                   o_refracted_ray_y,
                                   o_refracted_ray_z : real;
                                 var o_refracted_ray_flag : boolean );

var
  testKf,
  testKf2,
  Kf : real;
  Kn : real;
  abs_dot_product   : real;
  length_of_ray : real;
  v1_x, v1_y, v1_z  : real;

begin

{
  THIS PRODUCES THE ABSOLUTE VALUE OF THE DOT PRODUCT OF THE INCOMING RAY
 AND THE SURFACE NORMAL OF THE INTERSECTED SURFACE.
}
  abs_dot_product := Abs((i_obj_surface_normal_x * i_ray_vector_x) +
                         (i_obj_surface_normal_y * i_ray_vector_y) +
                         (i_obj_surface_normal_z * i_ray_vector_z));


  if abs_dot_product = 0 then  begin
{
  THIS IS JUST A PRECAUTION.
}
    o_refracted_ray_flag := false;
    o_refracted_ray_x := 0;
    o_refracted_ray_y := 0;
    o_refracted_ray_z := 0
   end
  else  begin
{
  THIS PRODUCES THE UNIT NORMAL VECTOR IN THE DIRECTION OF THE INCOMING
```

```
RAY.
}
    v1_x := i_ray_vector_x / abs_dot_product;
    v1_y := i_ray_vector_y / abs_dot_product;
    v1_z := i_ray_vector_z / abs_dot_product;

    length_of_ray := (sqrt(sqr(v1_x) + sqr(v1_y) + sqr(v1_z)));

{ THIS GIVES THE RATIO OF THE REFRACTIVE INDICES }
    Kn := i_obj_ridx / i_global_ridx;

{
  THE CALCULATION OF THE FRESNEL COEFFICIENT IS DIVIDED UP THIS WAY
 INORDER TO CHECK LATER FOR AN IMAGINARY DENOMINATOR WHICH INDICATES
 TOTAL INTERNAL REFLECTION.
}
    testKf  :=  (sqr(Kn) * sqr(length_of_ray));
    testKf2 :=  sqr(sqrt(sqr(i_obj_surface_normal_x + v1_x) +
                    sqr(i_obj_surface_normal_y + v1_y) +
                    sqr(i_obj_surface_normal_z + v1_z)));


    if (testKf - testKf2 <= 0 ) then  begin
{ IMAGINARY DENOMINATOR - TOTAL INTERNAL REFLECTION IS OCCURING }
      o_refracted_ray_flag := false;
      o_refracted_ray_x := 0;
      o_refracted_ray_y := 0;
      o_refracted_ray_z := 0
     end
    else  begin
      o_refracted_ray_flag := true;
      Kf := 1 / sqrt(testKf - testKf2);
      o_refracted_ray_x := (Kf * (i_obj_surface_normal_x + v1_x)) -
                    i_obj_surface_normal_x;
      o_refracted_ray_y := (Kf * (i_obj_surface_normal_y + v1_y)) -
                    i_obj_surface_normal_y;
      o_refracted_ray_z := (Kf * (i_obj_surface_normal_z + v1_z)) -
                    i_obj_surface_normal_z
    end
  end;

end; { calculate_refracted_ray }

{.PA}
{
 ********************** CALCULATE_REFLECTED_RAY **************************
 * CALLED FROM : MAIN and CALCULATE_INTENSITY
 * CALLS TO    : NONE
 * DESC    : CALCULATES THE DIRECTION OF A REFLECTED RAY
 * INPUT   :  Direction of the source ray and surface normal of intersected
 *         object.
 * OUTPUT  : Flag indicating existance of reflected ray. If one exists
```

```
*         then it's direction is given.
*************************************************************************
}
procedure calculate_reflected_ray (i_ray_vector_x,
                                   i_ray_vector_y,
                                   i_ray_vector_z       : real;
                                   i_obj_surface_normal_x,
                                   i_obj_surface_normal_y,
                                   i_obj_surface_normal_z : real;
                                  var o_reflected_ray_x,
                                   o_reflected_ray_y,
                                   o_reflected_ray_z : real:
                                  var o_reflected_ray_flag : boolean );

var
  abs_dot_product  : real;
  length_of_ray : real;
  v1_x, v1_y, v1_z  : real;
  R1_x, R1_y, R1_z  : real;

begin

{
 ABSOLUTE VALUE OF THE DOT PRODUCT OF THE OBJECTS SURFACE NORMAL AND
 OF THE INCOING LIGHT RAY.
}
  abs_dot_product := Abs((i_obj_surface_normal_x * i_ray_vector_x) +
                (i_obj_surface_normal_y * i_ray_vector_y) +
                (i_obj_surface_normal_z * i_ray_vector_z));

  v1_x := i_ray_vector_x / abs_dot_product;
  v1_y := i_ray_vector_y / abs_dot_product;
  v1_z := i_ray_vector_z / abs_dot_product;

{ CALCULATE UNIT NORMAL VECTOR IN THE DIRECTION OF THE INCOMING RAY. }
  R1_x := v1_x + (2 * i_obj_surface_normal_x);
  R1_y := v1_y + (2 * i_obj_surface_normal_y);
  R1_z := v1_z + (2 * i_obj_surface_normal_z);

  length_of_ray := (sqrt(sqr(R1_x) + sqr(R1_y) + sqr(R1_z)));

{ CALCULATE REFLECTED RAY }
  o_reflected_ray_x := R1_x / length_of_ray;
  o_reflected_ray_y := R1_y / length_of_ray;
  o_reflected_ray_z := R1_z / length_of_ray;
  o_reflected_ray_flag := true;

end; { calculate_reflected_ray }



{.PA}
```

```
{
*********************** CALCULATE_INTENSITY **********************
* CALLED FROM : I2 and I6
* CALLS TO    : CHECK_FOR_INTERSECTION and CALCULATE_REFLECTED_RAY
* DESC    :  Calculates the intensity at any given point.
* INPUT   :  Color component being calculated, current ray data,
*         pointer to object data, and pointer to light data.
* OUTPUT  :  Intensity at a given point, either to be displayed or
*         set in the appropriate source ray.
*********************************************************************
}
procedure CALCULATE_INTENSITY ( input_color        : colortype;
                        input_ray_d        : real;
                        input_ambient      : real;
                        input_ray_I_t      : real;
                        input_ray_I_s      : real;
                        input_ray_vector_x,
                        input_ray_vector_y,
                        input_ray_vector_z   : real;
                        input_number_of_lights  : integer;
                        input_obj_K_a      : real;
                        input_obj_K_s      : real;
                        input_obj_K_t      : real;
                        input_obj_K_d      : real;
                        input_obj_phong_exp  : integer;
                        input_intersection_x : real;
                        input_intersection_y : real;
                        input_intersection_z : real;
                        input_obj_surface_normal_x : real;
                        input_obj_surface_normal_y : real;
                        input_obj_surface_normal_z : real;
                        input_picture_object : object_ptr;
                        input_light_top  : light_ptr;
                    var io_intensity : real  );

var
  ans : char;
  I_d  : real;
  I_l  :real;
  j, i : integer;
  distance : real;
  sight_ray_x,
  sight_ray_y,
  sight_ray_z   : real;
  unit_sight_x,
  unit_sight_y,
  unit_sight_z  : real;
  light_ray_x,
  light_ray_y,
  light_ray_z   : real;
  unit_light_x,
  unit_light_y,
```

```
  unit_light_z  : real;
  unit_reflected_x,
  unit_reflected_y,
  unit_reflected_z  : real;
  intersection_x,
  intersection_y,
  intersection_z    : real;
  reflected_light_ray_x,
  reflected_light_ray_y,
  reflected_light_ray_z  : real;
  source_x,
  source_y,
  source_z  : real;
  check_x,
  check_y,
  check_z   : real;
  obj_idx, subobj_idx, cpart_idx, polygon_idx  : integer;
  intersection_flag : boolean;
  obj_light_distance : real;
  reflected_ray : boolean;
  intersected_obj_Kt : real;


begin

{ THIS ELIMINATES THE SAME INTERSECTION POINT FROM BEING SELECTED AGAIN.}
if input_ray_d > 0.1 then  begin
  I_l := 0.0;
  io_intensity := 0.0;
  reflected_ray := false;
  intersection_flag := false;

{ THIS SETS UP THE SIGHT RAY }
  sight_ray_x := -input_ray_vector_x;
  sight_ray_y := -input_ray_vector_y;
  sight_ray_z := -input_ray_vector_z;
  distance := sqrt(sqr(sight_ray_x) +
              sqr(sight_ray_y) +
              sqr(sight_ray_z));

  unit_sight_x := sight_ray_x / distance;
  unit_sight_y := sight_ray_y / distance;
  unit_sight_z := sight_ray_z / distance;

  for i := 1 to (input_number_of_lights + 1) do  begin
{ THIS GENERATES THE SHADOW FEELERS }

    light_ray_x := picture.lights^[i].light_x - input_intersection_x;
    light_ray_y := picture.lights^[i].light_y - input_intersection_y;
    light_ray_z := picture.lights^[i].light_z - input_intersection_z;
    distance := (sqrt(sqr(light_ray_x) +
                sqr(light_ray_y) +
```

```
                sqr(light_ray_z)));

{ CONVERTS IT TO A UNIT VECTOR }
    unit_light_x := light_ray_x / distance;
    unit_light_y := light_ray_y / distance;
    unit_light_z := light_ray_z / distance;

    source_x := picture.lights^[i].light_x;
    source_y := picture.lights^[i].light_y;
    source_z := picture.lights^[i].light_z;

{
  CHECK TO SEE IF ANY OF THE SHADOW FEELERS INTERSECT ANYTHING.
}
    check_for_intersection(-unit_light_x, -unit_light_y, -unit_light_z,
                    source_x,
                    source_y,
                    source_z,
                    input_picture_object,
                    intersection_x,
                    intersection_y,
                    intersection_z,
                    obj_idx, subobj_idx, cpart_idx, polygon_idx,
                    intersection_flag);

{
  CHECK TO INSURE THAT THE SAME POINT IS NOT CONSIDERED AGAIN, WHICH CAN
 HAPPEN.
}
    check_x := intersection_x - input_intersection_x;
    check_y := intersection_y - input_intersection_y;
    check_z := intersection_z - input_intersection_z;

{
  PULL THE PROPER CHARACTERISTICS OF THE OBJECT OUT TO DEAL WITH THE
 APPROPRIATE COMPONENT OF LIGHT THAT IS BEING CURRENTLY DEALT WITH
}
    if color = red then
      I_l := picture.lights^[i].I_r;
      intersected_obj_Kt := picture.
                    objects^[obj_idx].
                    sub_objects^[subobj_idx].
                    common_parts^[cpart_idx].
                    K_tr;
    if color = green then
      I_l := picture.lights^[i].I_g;
      intersected_obj_Kt := picture.
                    objects^[obj_idx].
                    sub_objects^[subobj_idx].
                    common_parts^[cpart_idx].
                    K_tg;
    if color = blue then
```

```
      I_l := picture.lights^[i].I_b;
      intersected_obj_Kt := picture.
                        objects^[obj_idx].
                        sub_objects^[subobj_idx].
                        common_parts^[cpart_idx].
                        K_tb;

    obj_light_distance := (sqrt(sqr(intersection_x -
                            picture.lights^[i].light_x) +
                          sqr(intersection_y -
                            picture.lights^[i].light_y) +
                          sqr(intersection_z -
                            picture.lights^[i].light_z)));

{
  IF THERE HAS BEEN AN INTERSETION AND THE POINT BEING CONSIDERED IS NOT
THE ORIGIN OF THE RAY THEN CHECK TO SEE IF THE OBJECT IS OPAQUE. IF IT IS
THEN RETURN TO BEGINNING OF LOOP AND CHECK NEXT SHADOW FEELER. IF IT IS
NOT OPAQUE THEN CALCULATE INTENSITY AT THAT POINT AND CONTINUE FOLLOWING
THE RAY TO SEE IF IT INTERSECTS ANYTHING ELSE. CONTINUE THIS LOOP
EITHER UNTIL NO MORE OBJECTS ARE LEFT OR UNTIL AN OPAQUE ONE IS
INTERSECTED.
}
{
  THIS PART OF THE CODE HAS NEVER BEEN TESTED. I JUST CODED IT AS I
THOUGHT IT SHOULD BE FROM THE ALGORITHM IN ROGERS BOOK PP. 377.
QUITE FRANKLY I STILL DON'T FULLY UNDERSTAND WHAT IS SUPPOSE TO TAKE PLACE
HERE.
}
    if intersection_flag then  begin
       while ((intersection_flag) and
            ((check_x > 0) or
             (check_y > 0) or
             (check_z > 0))) do  begin

         intersection_flag := false;

         if not( intersected_obj_Kt = 0) then  begin
           if (input_color = red)  then  begin

             I_l := picture.lights^[i].I_r *
                   intersected_obj_Kt:
            end;
           if (input_color = green)  then  begin

              I_l := picture.lights^[i].I_g  *
                    intersected_obj_Kt;
            end;
           if (input_color = blue) then begin

             I_l := picture.lights^[i].I_b  *
                   intersected_obj_Kt;
```

```
 end;

calculate_reflected_ray(input_ray_vector_x,
                        input_ray_vector_y,
                        input_ray_vector_z,
                        input_obj_surface_normal_x,
                        input_obj_surface_normal_y,
                        input_obj_surface_normal_z,
                        reflected_ray_x,
                        reflected_ray_y,
                        reflected_ray_z,
                        reflected_ray);

distance := sqrt(sqr(reflected_ray_x) +
                 sqr(reflected_ray_y) +
                 sqr(reflected_ray_z));

unit_reflected_x := reflected_ray_x / distance;
unit_reflected_y := reflected_ray_y / distance;
unit_reflected_z := reflected_ray_z / distance;


io_intensity := io_intensity +
                ((I_l * input_obj_K_d) *
                 ((input_obj_surface_normal_x * unit_light_x) +
                  (input_obj_surface_normal_y * unit_light_y) +
                  (input_obj_surface_normal_z * unit_light_z))) +
                ((I_l * input_obj_K_s) *
                 ((unit_sight_x * unit_reflected_x) +
                  (unit_sight_y * unit_reflected_y) +
                  (unit_sight_z * unit_reflected_z)));

source_x := intersection_x;
source_y := intersection_y;
source_z := intersection_z;

check_for_intersection(-unit_light_x,
                       -unit_light_y,
                       -unit_light_z,
                       source_x,
                       source_y,
                       source_z,
                       input_picture_object,
                       intersection_x,
                       intersection_y,
                       intersection_z,
                       obj_idx, subobj_idx,
                       cpart_idx, polygon_idx,
                       intersection_flag);

check_x := intersection_x - input_intersection_x;
check_y := intersection_y - input_intersection_y;
```

```
          check_z := intersection_z - input_intersection_z;

        end {if}
       end {while}
     end {if}
    else begin
      calculate_reflected_ray(input_ray_vector_x,
                      input_ray_vector_y,
                      input_ray_vector_z,
                      input_obj_surface_normal_x,
                      input_obj_surface_normal_y,
                      input_obj_surface_normal_z,
                      reflected_ray_x,
                      reflected_ray_y,
                      reflected_ray_z,
                      reflected_ray);

      distance := sqrt(sqr(reflected_ray_x) +
                  sqr(reflected_ray_y) +
                  sqr(reflected_ray_z));

      unit_reflected_x := reflected_ray_x / distance;
      unit_reflected_y := reflected_ray_y / distance;
      unit_reflected_z := reflected_ray_z / distance;

      io_intensity := io_intensity +
                 ((I_l * input_obj_K_d) *
                  ((input_obj_surface_normal_x * unit_light_x) +
                   (input_obj_surface_normal_y * unit_light_y) +
                   (input_obj_surface_normal_z * unit_light_z))) +
                 ((I_l * input_obj_K_s) *
                  ((unit_sight_x * unit_reflected_x) +
                  (unit_sight_y * unit_reflected_y) +
                  (unit_sight_z * unit_reflected_z)));
    end { ** ELSE **}
  end; { ** FOR ** }
{
 THIS IS THE STUB TO JUST HAVE EVERY OBJECT ILLUMINATED BY AMBIENT LIGHT
  io_intensity := input_obj_K_a * input_ambient;
}

{ ************************************************** }
{
  THIS IS WHERE THE FINAL INTENSITY IS CALCULATED
}
  io_intensity := ((input_obj_K_a * input_ambient) +
              io_intensity +
              ((input_obj_K_s * input_ray_I_s) +
               (input_obj_K_t * input_ray_I_t))) ;
                              {/input_ray_d or /2 or /1}

{
```

```
  THIS IS JUST TO KEEP ALL VALUES WITHIN A RANGE WHERE THEY CAN BE DISPLAYED
}
  if io_intensity > 1.00 then
    io_intensity := 1.00;
  if io_intensity < 0.00 then
    io_intensity := 0.00;

{ ********************************************************** }
 end
else  begin
{
 IF THE INPUT RAY DISTANCE IS LESS THEN ONE THAN YOU ARE CONSIDERING THE
 SAME POINT AND HENCE THE INTENSITY THERE SHOULD BE 0.
}
  io_intensity := 0.0
end;

end; { CALCULATE_INTENSITY }

{.PA}
{
 ************************ DISPLAY_PIXEL ****************************
 * CALLED FROM : MAIN
 * CALLS TO : NONE
 * DESC :  WRITES OUTPUT TO FILE
 * INPUT : THE VALUES FOR THE RED, GREEN, AND BLUE COMPONENTS OF LIGHT.
 * OUTPUT : NONE
 ***********************************************************************
}
procedure DISPLAY_PIXEL ( input_intensity_red,
                  input_intensity_green,
                  input_intensity_blue   : real;
                  input_pixel_x,
                  input_pixel_y,
                  input_pixel_z          : real );

begin

  write (outfile,input_intensity_red:3:2);
  write (outfile,' ',input_intensity_green:3:2);
  writeln (outfile,' ',input_intensity_blue:3:2);

end; { DISPLAY_PIXEL }

{.PA}
{
 ************************** POP ************************************
 * CALLED FROM : MAIN, I2, I6
 * CALLS TO : NONE
 * DESC : Removes a ray from the top of the stack.
 * INPUT : Pointer to the current top top of stack.
 * OUTPUT : The ray just popped from the stack and a pointer to the new top
```

```
*       top of stack.
*****************************************************************************
}
procedure POP (var output_ray_type        : raytype;
          var output_ray_origin_x      : real;
          var output_ray_origin_y      : real;
          var output_ray_origin_z      : real;
          var output_ray_vector_x      : real;
          var output_ray_vector_y      : real;
          var output_ray_vector_z      : real;
          var output_ray_stype        : raytype;
          var output_intersection_flag : boolean;
          var output_obj_idx          : integer;
          var output_subobj_idx       : integer;
          var output_cpart_idx        : integer;
          var output_polygon_idx      : integer;
          var output_intersection_x    : real;
          var output_intersection_y    : real;
          var output_intersection_z    : real;
          var output_d                : real;
          var output_I_tr, output_I_tg, output_I_tb  : real;
          var output_I_sr, output_I_sg, output_I_sb  : real;
          var io_top                  : ray_ptr      );

begin

  if (stack_empty (io_top)) then  begin
    writeln('STACK UNDERFLOW ERROR')
   end { if }
  else  begin
    output_ray_type         := io_top^.ray_type;
    output_ray_origin_x      := io_top^.ray_origin_x;
    output_ray_origin_y      := io_top^.ray_origin_y;
    output_ray_origin_z      := io_top^.ray_origin_z;
    output_ray_vector_x      := io_top^.ray_vector_x;
    output_ray_vector_y      := io_top^.ray_vector_y;
    output_ray_vector_z      := io_top^.ray_vector_z;
    output_ray_stype        := io_top^.ray_stype;
    output_intersection_flag  := io_top^.intersection_flag;
    output_obj_idx          := io_top^.obj_idx;
    output_subobj_idx        := io_top^.subobj_idx;
    output_cpart_idx         := io_top^.cpart_idx;
    output_polygon_idx       := io_top^.polygon_idx;
    output_intersection_x    := io_top^.intersection_x;
    output_intersection_y    := io_top^.intersection_y;
    output_intersection_z    := io_top^.intersection_z;
    output_d                := io_top^.d;
    output_I_tr             := io_top^.I_tr;
    output_I_tg             := io_top^.I_tg;
    output_I_tb             := io_top^.I_tb;
    output_I_sr             := io_top^.I_sr;
    output_I_sg             := io_top^.I_sg;
```

```
    output_I_sb          := io_top^.I_sb;
    ray_next             := io_top^.ray_link;
{ remove old pointer }
    dispose(io_top);
{ set top of stack pointer to new top of stack }
    io_top := ray_next;
  end; { else }
end; { POP }

{.PA}
{
*************************** PUSH *********************************
* CALLED FROM: MAIN, I2, I6
* CALLS TO : NONE
* DESC : Places a ray on the top of the stack.
* INPUT: The current top of the stack, and the data for a new ray
* OUTPUT: The pointer to th new top of stack.
*******************************************************************
}
procedure PUSH ( input_ray_type       : raytype;
            input_ray_origin_x    : real;
            input_ray_origin_y    : real;
            input_ray_origin_z    : real;
            input_ray_vector_x    : real;
            input_ray_vector_y    : real;
            input_ray_vector_z    : real;
            input_ray_stype       : raytype;
            input_intersection_flag : boolean;
            input_obj_idx         : integer;
            input_subobj_idx      : integer;
            input_cpart_idx       : integer;
            input_polygon_idx     : integer;
            input_intersection_x  : real;
            input_intersection_y  : real;
            input_intersection_z  : real;
            input_d               : real;
            input_I_tr, input_I_tg, input_I_tb : real;
            input_I_sr, input_I_sg, input_I_sb : real;
         var io_top               : ray_ptr );

begin

  new(ray_current);
  ray_current^.ray_type        := input_ray_type;
  ray_current^.ray_origin_x    := input_ray_origin_x;
  ray_current^.ray_origin_y    := input_ray_origin_y;
  ray_current^.ray_origin_z    := input_ray_origin_z;
  ray_current^.ray_vector_x    := input_ray_vector_x;
  ray_current^.ray_vector_y    := input_ray_vector_y;
  ray_current^.ray_vector_z    := input_ray_vector_z;
  ray_current^.ray_stype       := input_ray_stype;
  ray_current^.intersection_flag := input_intersection_flag;
```

```
  ray_current^.obj_idx          := input_obj_idx;
  ray_current^.subobj_idx       := input_subobj_idx;
  ray_current^.cpart_idx        := input_cpart_idx;
  ray_current^.polygon_idx      := input_polygon_idx;
  ray_current^.intersection_x   := input_intersection_x;
  ray_current^.intersection_y   := input_intersection_y;
  ray_current^.intersection_z   := input_intersection_z;
  ray_current^.d                := input_d;
  ray_current^.I_tr             := input_I_tr;
  ray_current^.I_tg             := input_I_tg;
  ray_current^.I_tb             := input_I_tb;
  ray_current^.I_sr             := input_I_sr;
  ray_current^.I_sg             := input_I_sg;
  ray_current^.I_sb             := input_I_sb;
  ray_current^.ray_link         := io_top;
  io_top                    := ray_current;

end; { PUSH }
```

```
PROGRAM RAYTRACER;
{
 ************************************************************************
 ************************************************************************
 ** PROG   : RAY.PAS
 ** AUTHOR : Paul G. Smith
 ** DATE   : 11 May 1987
 ** DESC   :  A ray tracing prototype with a global illumination model
 **         integrated into it.
 ** INPUT  :  A sequential scene file under PICTURE.PAS.
 ** OUTPUT :  A bitmap file containing the red, green, and blue color
 **         color components. Their values range from 0-1 and need to
 **         be converted for display on an RGB color monitor.
 ************************************************************************
 ************************************************************************
}

{ * INCLUDE FILES   * }

{$I declare6.pas} - { DECLARATION SECTION }
{.PA}
{$I intprcs6.pas} - { INTERSECTION PROCEDURES }
{.PA}
{$I procs6.pas}   _ { INTENSITY and UTILITY PROCEDURES }

{.PA}
{
 ********************************** I2 *********************************
 * CALLED FROM: MAIN
 * CALLS TO:    PUSH, POP, CALCULATE_INTENSITY
 * DESC : Calculates the light intensity at a given intersection point
 * INPUT : A complete ray data record, the pointer to the light array
 *       and the pointer to the object array
 * OUTPUT : The intensity at a given intersection point. If the input
 *        was a view ray then this intensity will be the intensity
 *        displayed. If the input ray is a reflected ray then this
 *        intensity is assigned to the I_s field in the source ray.
 *        If the input ray is a refracted ray then this intensity is
 *        assigned to the I_t field in the source ray.
 ************************************************************************
}

procedure I2 (i_d     : real;
          i_I_tr, i_I_tg, i_I_tb    : real;
          i_I_sr, i_I_sg, i_I_sb    : real;
          i_number_of_light_sources : integer;
          i_ambient_r, i_ambient_g, i_ambient_b    : real;
          i_K_ar, i_K_ag, i_K_ab    : real;
          i_K_sr, i_K_sg, i_K_sb    : real;
```

```
        i_K_tr, i_K_tg, i_K_tb    : real;
        i_K_dr, i_K_dg, i_K_db    : real;
        i_ray_type  : raytype;
        i_ray_top   : ray_ptr;
        i_light_top : light_ptr;
        i_obj_ptr   : object_ptr;
        i_obj_phong_exp : integer;
        i_ray_vector_x,
        i_ray_vector_y,
        i_ray_vector_z          : real;
        i_intersection_x,
        i_intersection_y,
        i_intersection_z        : real;
        i_surface_normal_x,
        i_surface_normal_y,
        i_surface_normal_z      : real;
      var io_ray_generation_number : integer;
      var o_intensity_red,
        o_intensity_green,
        o_intensity_blue  : real );

var
{ SET UP TEMPORARY AREA TO HOLD RAYS POPPED FROM STACK }
  temp1_ray_type   : raytype;
  temp1_ray_origin_x,
  temp1_ray_origin_y,
  temp1_ray_origin_z : real;
  temp1_ray_vector_x,
  temp1_ray_vector_y,
  temp1_ray_vector_z : real;
  temp1_ray_stype    : raytype;
  temp1_intersection_flag : boolean;
  temp1_obj_idx,
  temp1_subobj_idx,
  temp1_cpart_idx,
  temp1_polygon_idx       : integer;
  temp1_intersection_x,
  temp1_intersection_y,
  temp1_intersection_z    : real;
  temp1_d    : real;
  temp1_I_tr, temp1_I_tg, temp1_I_tb : real;
  temp1_I_sr, temp1_I_sg, temp1_I_sb : real;

  temp2_ray_type   : raytype;
  temp2_ray_origin_x,
  temp2_ray_origin_y,
  temp2_ray_origin_z : real;
  temp2_ray_vector_x,
  temp2_ray_vector_y,
  temp2_ray_vector_z : real;
  temp2_ray_stype    : raytype;
  temp2_intersection_flag  : boolean;
```

```
  temp2_obj_idx,
  temp2_subobj_idx,
  temp2_cpart_idx,
  temp2_polygon_idx       : integer;
  temp2_intersection_x,
  temp2_intersection_y,
  temp2_intersection_z    : real;
  temp2_d   : real;
  temp2_I_tr, temp2_I_tg, temp2_I_tb : real;
  temp2_I_sr, temp2_I_sg, temp2_I_sb : real;

begin

{ CALCULATE INTENSITY OF THE RED COMPONENT OF LIGHT }
  calculate_intensity( red,
                i_d,
                i_ambient_r,
                i_I_tr,
                i_I_sr,
                i_ray_vector_x, i_ray_vector_y, i_ray_vector_z,
                i_number_of_light_sources,
                i_K_ar,
                i_K_sr,
                i_K_tr,
                i_K_dr,
                i_obj_phong_exp,
                i_intersection_x,
                i_intersection_y,
                i_intersection_z,
                i_surface_normal_x,
                i_surface_normal_y,
                i_surface_normal_z,
                i_obj_ptr,
                i_light_top,
                o_intensity_red   );

{ CALCULATE INTENSITY OF THE GREEN COMPONENT OF LIGHT }
  calculate_intensity( green,
                i_d,
                i_ambient_g,
                i_I_tg,
                i_I_sg,
                i_ray_vector_x, i_ray_vector_y, i_ray_vector_z,
                i_number_of_light_sources,
                i_K_ag,
                i_K_sg,
                i_K_tg,
                i_K_dg,
                i_obj_phong_exp,
                i_intersection_x,
                i_intersection_y,
                i_intersection_z,
```

```
                i_surface_normal_x,
                i_surface_normal_y,
                i_surface_normal_z,
                i_obj_ptr,
                i_light_top,
                o_intensity_green );

{ CALCULATE INTENSITY OF THE BLUE COMPONENT OF LIGHT }
  calculate_intensity( blue,
                i_d,
                i_ambient_b,
                i_I_tb,
                i_I_sb,
                i_ray_vector_x, i_ray_vector_y, i_ray_vector_z,
                i_number_of_light_sources,
                i_K_ab,
                i_K_sb,
                i_K_tb,
                i_K_db,
                i_obj_phong_exp,
                i_intersection_x,
                i_intersection_y,
                i_intersection_z,
                i_surface_normal_x,
                i_surface_normal_y,
                i_surface_normal_z,
                i_obj_ptr,
                i_light_top,
                o_intensity_blue );

  if (i_ray_type = view)  then begin
{
  THE VIEW RAY IS ALWAYS THE LAST RAY ON THE STACK WHEN IT IS POPPED THE
 INTENSITY DETERMINED FOR IT IS PASSED BACK INTO MAIN FOR DISPLAY.
}

      { nothing }
    end
  else begin

    if (i_ray_type = reflected)  then begin
{
  SINCE THIS IS THE REFLECTED RAY THEN JUST ONE RAY NEEDS TO BE POPPED TO
 GAIN ACCESS TO THE SOURCE RAY.
}
      pop (temp1_ray_type,
          temp1_ray_origin_x, temp1_ray_origin_y, temp1_ray_origin_z,
          temp1_ray_vector_x, temp1_ray_vector_y, temp1_ray_vector_z,
          temp1_ray_stype,
          temp1_intersection_flag,
          temp1_obj_idx,
          temp1_subobj_idx,
```

```
        temp1_cpart_idx,
        temp1_polygon_idx,
        temp1_intersection_x, temp1_intersection_y, temp1_intersection_z,
        temp1_d,
        temp1_I_tr, temp1_I_tg, temp1_I_tb,
        temp1_I_sr, temp1_I_sg, temp1_I_sb,
        ray_top      );

{ SET INTENSITY IN SOURCE RAY }
    temp1_I_sr := o_intensity_red;
    temp1_I_sg := o_intensity_green;
    temp1_I_sb := o_intensity_blue;

{ RESTORE STACK }
    push( temp1_ray_type,
        temp1_ray_origin_x, temp1_ray_origin_y, temp1_ray_origin_z,
        temp1_ray_vector_x, temp1_ray_vector_y, temp1_ray_vector_z,
        temp1_ray_stype,
        temp1_intersection_flag,
        temp1_obj_idx,
        temp1_subobj_idx,
        temp1_cpart_idx,
        temp1_polygon_idx,
        temp1_intersection_x, temp1_intersection_y, temp1_intersection_z,
        temp1_d,
        temp1_I_tr, temp1_I_tg, temp1_I_tb,
        temp1_I_sr, temp1_I_sg, temp1_I_sb,
        ray_top          );

   end
   else begin
{
  SINCE THIS IS THE REFRACTED RAY TWO RAYS MUST BE POPPED TO GAIN ACCESS
 TO THE SOURCE RAY.
}
    pop (temp1_ray_type,
        temp1_ray_origin_x, temp1_ray_origin_y, temp1_ray_origin_z,
        temp1_ray_vector_x, temp1_ray_vector_y, temp1_ray_vector_z,
        temp1_ray_stype,
        temp1_intersection_flag,
        temp1_obj_idx,
        temp1_subobj_idx,
        temp1_cpart_idx,
        temp1_polygon_idx,
        temp1_intersection_x, temp1_intersection_y, temp1_intersection_z,
        temp1_d,
        temp1_I_tr, temp1_I_tg, temp1_I_tb,
        temp1_I_sr, temp1_I_sg, temp1_I_sb,
        ray_top      );

    pop (temp2_ray_type,
        temp2_ray_origin_x, temp2_ray_origin_y, temp2_ray_origin_z,
```

```
        temp2_ray_vector_x, temp2_ray_vector_y, temp2_ray_vector_z,
        temp2_ray_stype,
        temp2_intersection_flag,
        temp2_obj_idx,
        temp2_subobj_idx,
        temp2_cpart_idx,
        temp2_polygon_idx,
        temp2_intersection_x, temp2_intersection_y, temp2_intersection_z,
        temp2_d,
        temp2_I_tr, temp2_I_tg, temp2_I_tb,
        temp2_I_sr, temp2_I_sg, temp2_I_sb,
        ray_top      );

{ SET INTENSITY IN THE SOURCE RAY. }
    temp2_I_tr := o_intensity_red;
    temp2_I_tg := o_intensity_green;
    temp2_I_tb := o_intensity_blue;

{ RESTORE STACK. }
    push( temp2_ray_type,
        temp2_ray_origin_x, temp2_ray_origin_y, temp2_ray_origin_z,
        temp2_ray_vector_x, temp2_ray_vector_y, temp2_ray_vector_z,
        temp2_ray_stype,
        temp2_intersection_flag,
        temp2_obj_idx,
        temp2_subobj_idx,
        temp2_cpart_idx,
        temp2_polygon_idx,
        temp2_intersection_x, temp2_intersection_y, temp2_intersection_z,
        temp2_d,
        temp2_I_tr, temp2_I_tg, temp2_I_tb,
        temp2_I_sr, temp2_I_sg, temp2_I_sb,
        ray_top          );

    push( temp1_ray_type,
        temp1_ray_origin_x, temp1_ray_origin_y, temp1_ray_origin_z,
        temp1_ray_vector_x, temp1_ray_vector_y, temp1_ray_vector_z,
        temp1_ray_stype,
        temp1_intersection_flag,
        temp1_obj_idx,
        temp1_subobj_idx,
        temp1_cpart_idx,
        temp1_polygon_idx,
        temp1_intersection_x, temp1_intersection_y, temp1_intersection_z,
        temp1_d,
        temp1_I_tr, temp1_I_tg, temp1_I_tb,
        temp1_I_sr, temp1_I_sg, temp1_I_sb,
        ray_top          );

    end;
{
  SINCE THE RAY SENT INTO THIS PROCEDURE IS NO LONGER NEEDED IT IS
```

```
 DISCARDED, HENCE THE RAY COUNT NEEDS TO BE DECREMENTED.
}
    io_ray_generation_number := io_ray_generation_number - 1;
  end;
end;

{.PA}
{
********************************* I6 ***********************************
* CALLED FROM : MAIN
* CALLS TO    : PUSH, POP, CALCULATE_INTENSITY
* DESC   : Used to calculate intensity at node of stack storage is
*       exceeded.
* INPUT  : Incomplete ray data record, this ray can not be continued
*       because there is no room on the stack for it. Also pointers
*       to the light array and object array.
* OUTPUT : Intensity, I_t or I_s, is set in source ray.
************************************************************************
}

procedure I6 (i_d                    : real;
          i_I_tr, i_I_tg, i_I_tb   : real;
          i_I_sr, i_I_sg, i_I_sb   : real;
          i_number_of_light_sources : integer;
          i_ambient_r, i_ambient_g, i_ambient_b : real;
          i_K_ar, i_K_ag, i_K_ab   : real;
          i_K_sr, i_K_sg, i_K_sb   : real;
          i_K_tr, i_K_tg, i_K_tb   : real;
          i_K_dr, i_K_dg, i_K_db   : real;
          i_ray_type     : raytype;
          i_ray_top      : ray_ptr;
          i_light_top    : light_ptr;
          i_obj_ptr      : object_ptr;
          i_obj_phong_exp : integer;
          i_ray_vector_x,
          i_ray_vector_y,
          i_ray_vector_z      : real;
          i_intersection_x,
          i_intersection_y,
          i_intersection_z    : real;
          i_surface_normal_x,
          i_surface_normal_y,
          i_surface_normal_z  : real;
        var o_intensity_red,
          o_intensity_green,
          o_intensity_blue    : real );

var
{ STORAGE FOR POPPED RAYS }
  temp1_ray_type   : raytype;
  temp1_ray_origin_x,
  temp1_ray_origin_y,
```

```
  temp1_ray_origin_z : real;
  temp1_ray_vector_x,
  temp1_ray_vector_y,
  temp1_ray_vector_z : real;
  temp1_ray_stype    : raytype;
  temp1_intersection_flag  : boolean;
  temp1_obj_idx,
  temp1_subobj_idx,
  temp1_cpart_idx,
  temp1_polygon_idx     : integer;
  temp1_intersection_x,
  temp1_intersection_y,
  temp1_intersection_z   : real;
  temp1_d    : real;
  temp1_I_tr, temp1_I_tg, temp1_I_tb : real;
  temp1_I_sr, temp1_I_sg, temp1_I_sb : real;

  temp2_ray_type   : raytype;
  temp2_ray_origin_x,
  temp2_ray_origin_y,
  temp2_ray_origin_z : real;
  temp2_ray_vector_x,
  temp2_ray_vector_y,
  temp2_ray_vector_z : real;
  temp2_ray_stype    : raytype;
  temp2_intersection_flag  : boolean;
  temp2_obj_idx,
  temp2_subobj_idx,
  temp2_cpart_idx,
  temp2_polygon_idx       : integer;
  temp2_intersection_x,
  temp2_intersection_y,
  temp2_intersection_z    : real;
  temp2_d    : real;
  temp2_I_tr, temp2_I_tg, temp2_I_tb : real;
  temp2_I_sr, temp2_I_sg, temp2_I_sb : real;

begin

  {*** COULD INSERT TREE EXTENSION PROCEDURE ***}

{ CALCULATE RED COMPONENT OF LIGHT }
  calculate_intensity( red,
                       i_d,
                       i_ambient_r,
                       i_I_tr,
                       i_I_sr,
                       i_ray_vector_x, i_ray_vector_y, i_ray_vector_z,
                       i_number_of_light_sources,
                       i_K_ar,
                       i_K_sr,
                       i_K_tr,
```

```
                i_K_dr,
                i_obj_phong_exp,
                i_intersection_x,
                i_intersection_y,
                i_intersection_z,
                i_surface_normal_x,
                i_surface_normal_y,
                i_surface_normal_z,
                i_obj_ptr,
                i_light_top,
                o_intensity_red );

{ CALCULATE GREEN COMPONENT OF LIGHT }
  calculate_intensity( green,
                i_d,
                i_ambient_g,
                i_I_tg,
                i_I_sg,
                i_ray_vector_x, i_ray_vector_y, i_ray_vector_z,
                i_number_of_light_sources,
                i_K_ag,
                i_K_sg,
                i_K_tg,
                i_K_dg,
                i_obj_phong_exp,
                i_intersection_x,
                i_intersection_y,
                i_intersection_z,
                i_surface_normal_x,
                i_surface_normal_y,
                i_surface_normal_z,
                i_obj_ptr,
                i_light_top,
                o_intensity_green);

{ CALCULATE BLUE COMPONENT OF LIGHT }
  calculate_intensity( blue,
                i_d,
                i_ambient_b,
                i_I_tb,
                i_I_sb,
                i_ray_vector_x, i_ray_vector_y, i_ray_vector_z,
                i_number_of_light_sources,
                i_K_ab,
                i_K_sb,
                i_K_tb,
                i_K_db,
                i_obj_phong_exp,
                i_intersection_x,
                i_intersection_y,
                i_intersection_z,
                i_surface_normal_x,
```

```
                i_surface_normal_y,
                i_surface_normal_z,
                i_obj_ptr,
                i_light_top,
                o_intensity_blue);


 if (i_ray_type = reflected) then begin
{
  IF INPUT RAY IS A REFLECTED RAY THEN ONE RAY MUST BE POPPED TO GAIN
 ACCESS TO IT'S SOURCE RAY.
}
   pop (temp1_ray_type,
      temp1_ray_origin_x, temp1_ray_origin_y, temp1_ray_origin_z,
      temp1_ray_vector_x, temp1_ray_vector_y, temp1_ray_vector_z,
      temp1_ray_stype,
      temp1_intersection_flag,
      temp1_obj_idx,
      temp1_subobj_idx,
      temp1_cpart_idx,
      temp1_polygon_idx,
      temp1_intersection_x, temp1_intersection_y, temp1_intersection_z,
      temp1_d,
      temp1_I_tr, temp1_I_tg, temp1_I_tb,
      temp1_I_sr, temp1_I_sg, temp1_I_sb,
      i_ray_top      );

{ SET I_s IN SOURCE RAY }
    temp1_I_sr := o_intensity_red;
    temp1_I_sg := o_intensity_green;
    temp1_I_sb := o_intensity_blue;

{ RETORE STACK. }
    push( temp1_ray_type,
        temp1_ray_origin_x, temp1_ray_origin_y, temp1_ray_origin_z,
        temp1_ray_vector_x, temp1_ray_vector_y, temp1_ray_vector_z,
        temp1_ray_stype,
        temp1_intersection_flag,
        temp1_obj_idx,
        temp1_subobj_idx,
        temp1_cpart_idx,
        temp1_polygon_idx,
        temp1_intersection_x, temp1_intersection_y, temp1_intersection_z,
        temp1_d,
        temp1_I_tr, temp1_I_tg, temp1_I_tb,
        temp1_I_sr, temp1_I_sg, temp1_I_sb,
        ray_top          );


  end
 else begin
{
```

```
  IF INPUT RAY IS A REFRACTED RAY THEN TWO RAYS MUST BE POPPED FROM THE
STACK TO GAIN ACCESS TO IT'S SOURCE RAY.
}
    pop (temp1_ray_type,
        temp1_ray_origin_x, temp1_ray_origin_y, temp1_ray_origin_z,
        temp1_ray_vector_x, temp1_ray_vector_y, temp1_ray_vector_z,
        temp1_ray_stype,
        temp1_intersection_flag,
        temp1_obj_idx,
        temp1_subobj_idx,
        temp1_cpart_idx,
        temp1_polygon_idx,
        temp1_intersection_x, temp1_intersection_y, temp1_intersection_z,
        temp1_d,
        temp1_I_tr, temp1_I_tg, temp1_I_tb,
        temp1_I_sr, temp1_I_sg, temp1_I_sb,
        ray_top      );

    pop (temp2_ray_type,
        temp2_ray_origin_x, temp2_ray_origin_y, temp2_ray_origin_z,
        temp2_ray_vector_x, temp2_ray_vector_y, temp2_ray_vector_z,
        temp2_ray_stype,
        temp2_intersection_flag,
        temp2_obj_idx,
        temp2_subobj_idx,
        temp2_cpart_idx,
        temp2_polygon_idx,
        temp2_intersection_x, temp2_intersection_y, temp2_intersection_z,
        temp2_d,
        temp2_I_tr, temp2_I_tg, temp2_I_tb,
        temp2_I_sr, temp2_I_sg, temp2_I_sb,
        ray_top      );

{ SET THE I_t FIELD IN THE SOURCE RAY }
    temp2_I_tr := o_intensity_red;
    temp2_I_tg := o_intensity_green;
    temp2_I_tb := o_intensity_blue;

{ RESTORE THE STACK }
    push( temp2_ray_type,
         temp2_ray_origin_x, temp2_ray_origin_y, temp2_ray_origin_z,
         temp2_ray_vector_x, temp2_ray_vector_y, temp2_ray_vector_z,
         temp2_ray_stype,
         temp2_intersection_flag,
         temp2_obj_idx,
         temp2_subobj_idx,
         temp2_cpart_idx,
         temp2_polygon_idx,
         temp2_intersection_x, temp2_intersection_y, temp2_intersection_z,
         temp2_d,
         temp2_I_tr, temp2_I_tg, temp2_I_tb,
         temp2_I_sr, temp2_I_sg, temp2_I_sb,
```

```
        ray_top         );

    push( templ_ray_type,
        templ_ray_origin_x, templ_ray_origin_y, templ_ray_origin_z,
        templ_ray_origin_x, templ_ray_origin_y, templ_ray_origin_z,
        templ_ray_stype,
        templ_intersection_flag,
        templ_obj_idx,
        templ_subobj_idx,
        templ_cpart_idx,
        templ_polygon_idx,
        templ_intersection_x, templ_intersection_y, templ_intersection_z,
        templ_d,
        templ_I_tr, templ_I_tg, templ_I_tb,
        templ_I_sr, templ_I_sg, templ_I_sb,
        ray_top         );
  end;

end;


{.PA}
{ ************************************************************************** }
{ ********************************* MAIN ********************************** }
{ ************************************************************************** }
begin
{ INPUT FILE }
  assign (sysin, 'picture5.pas');
  reset  (sysin);

{ OUTPUT FILE }
  assign (outfile, 'pic0.dta');
  rewrite (outfile);


{ ************************************************************************** }

{ SET UP COUNTERS }
  obj_cntr    := 1;
  subobj_cntr := 1;
  light_cntr  := 1;
  cpart_cntr  := 1;
  poly_cntr   := 1;
  vertice_cntr := 1;

{ CREATE POINTERS TO RECORDS }
  new(obj_curr);
  new(subobj_curr);
  new(cpart_curr);
  new(light_current);
  new(poly_curr);
```

```
{ *** READ IN DATA FILE *** }

{ READ IN PICTURE RECORD }
  readln (sysin, picture.view_position_x);
  readln (sysin, picture.view_position_y);
  readln (sysin, picture.view_position_z);
  readln (sysin, picture.background_color_r);
  readln (sysin, picture.background_color_g);
  readln (sysin, picture.background_color_b);
  readln (sysin, picture.screen_max_x);
  readln (sysin, picture.screen_max_y);
  readln (sysin, picture.ambient_r);
  readln (sysin, picture.ambient_g);
  readln (sysin, picture.ambient_b);
  readln (sysin, picture.no_zero);
  readln (sysin, picture.global_refraction_index);
  readln (sysin, picture.num_lights);
  picture.lights := light_current;

  while picture.num_lights > 0 do begin

{ READ IN LIGHT DATA }
    readln (sysin,light_current^[light_cntr].I_r);
    readln (sysin,light_current^[light_cntr].I_g);
    readln (sysin,light_current^[light_cntr].I_b);
    readln (sysin,light_current^[light_cntr].light_x);
    readln (sysin,light_current^[light_cntr].light_y);
    readln (sysin,light_current^[light_cntr].light_z);
    readln (sysin,light_current^[light_cntr].dimension1);
    readln (sysin,light_current^[light_cntr].dimension2);
    light_cntr := light_cntr + 1;
    picture.num_lights := picture.num_lights - 1;
  end;
  readln (sysin, picture.num_objs);
  object_loop_cnt := picture.num_objs;
  picture.objects := obj_curr;


  while object_loop_cnt > 0 do  begin
{ READ IN OBJECT DATA }
    readln (sysin, obj_curr^[obj_cntr].opcode);
    readln (sysin, obj_curr^[obj_cntr].obj_bsphere_radius);
    readln (sysin, obj_curr^[obj_cntr].obj_bsphere_x);
    readln (sysin, obj_curr^[obj_cntr].obj_bsphere_y);
    readln (sysin, obj_curr^[obj_cntr].obj_bsphere_z);
    readln (sysin, obj_curr^[obj_cntr].num_sub_objects);
    subobj_loop_cnt := obj_curr^[obj_cntr].num_sub_objects;
    picture.objects^[obj_cntr].sub_objects := subobj_curr;

    while subobj_loop_cnt > 0 do  begin
{ READ IN SUBOBJECT DATA }
      readln (sysin, subobj_curr^[subobj_cntr].subobj_type);
```

```
        readln (sysin, subobj_curr^[subobj_cntr].sub_bsphere_radius);
        readln (sysin, subobj_curr^[subobj_cntr].sub_bsphere_x);
        readln (sysin, subobj_curr^[subobj_cntr].sub_bsphere_y);
        readln (sysin, subobj_curr^[subobj_cntr].sub_bsphere_z);
        readln (sysin, subobj_curr^[subobj_cntr].num_common_parts);
        cpart_loop_cnt := subobj_curr^[subobj_cntr].num_common_parts;
        picture.objects^[obj_cntr].sub_objects^[subobj_cntr].common_parts :=
          cpart_curr;

        while cpart_loop_cnt > 0 do  begin
{ READ IN COMMON PART DATA }
          readln(sysin, cpart_curr^[cpart_cntr].K_ar);
          readln(sysin, cpart_curr^[cpart_cntr].K_ag);
          readln(sysin, cpart_curr^[cpart_cntr].K_ab);
          readln(sysin, cpart_curr^[cpart_cntr].K_dr);
          readln(sysin, cpart_curr^[cpart_cntr].K_dg);
          readln(sysin, cpart_curr^[cpart_cntr].K_db);
          readln(sysin, cpart_curr^[cpart_cntr].K_sr);
          readln(sysin, cpart_curr^[cpart_cntr].K_sg);
          readln(sysin, cpart_curr^[cpart_cntr].K_sb);
          readln(sysin, cpart_curr^[cpart_cntr].K_tr);
          readln(sysin, cpart_curr^[cpart_cntr].K_tg);
          readln(sysin, cpart_curr^[cpart_cntr].K_tb);
          readln(sysin, cpart_curr^[cpart_cntr].obj_refraction_index);
          readln(sysin, cpart_curr^[cpart_cntr].obj_phong_exp);

{ CHECK TO SEE IF SUBOBJECT IS A SPHERE OR A POLYGONAL OBJECT }
          if (subobj_curr^[subobj_cntr].subobj_type = 1) then begin

            readln(sysin, cpart_curr^[cpart_cntr].num_polygons);
            poly_loop_cnt := cpart_curr^[cpart_cntr].num_polygons;
            picture.objects^[obj_cntr].sub_objects^[subobj_cntr].
              common_parts^[cpart_cntr].polygons := poly_curr;

            while poly_loop_cnt > 0 do  begin
{ READ IN POLYGON DATA }
              readln(sysin, poly_curr^[poly_cntr].num_vertices);
              vertice_loop_cnt := poly_curr^[poly_cntr].num_vertices;

              while vertice_loop_cnt > 0 do begin
{ READ IN VERTICE DATA }
                readln(sysin, poly_curr^[poly_cntr].vertice_x[vertice_cntr]);
                readln(sysin, poly_curr^[poly_cntr].vertice_y[vertice_cntr]);
                readln(sysin, poly_curr^[poly_cntr].vertice_z[vertice_cntr]);
                vertice_cntr := vertice_cntr + 1;
                vertice_loop_cnt := vertice_loop_cnt - 1;
              end;
              vertice_cntr := 1;
              readln(sysin, poly_curr^[poly_cntr].surface_normal_x);
              readln(sysin, poly_curr^[poly_cntr].surface_normal_y);
              readln(sysin, poly_curr^[poly_cntr].surface_normal_z);
              poly_cntr := poly_cntr + 1;
```

```
            poly_loop_cnt := poly_loop_cnt - 1

          end;
          poly_cntr := 1;
          new(poly_curr);

        end; { IF }
        cpart_cntr := cpart_cntr + 1;
        cpart_loop_cnt := cpart_loop_cnt - 1

      end;
      cpart_cntr := 1;
      new(cpart_curr);
      subobj_cntr := subobj_cntr + 1;
      subobj_loop_cnt := subobj_loop_cnt - 1

    end;
    new(subobj_curr);
    subobj_cntr := 1;
    obj_cntr := obj_cntr + 1;
    object_loop_cnt  := object_loop_cnt - 1

  end;

{.PA}
{ ************************************************************************** }
{ SET RAY STACK POINTER }
  ray_top    := nil;

  pixel_z := initial_pixel_z;

{ RASTER SCAN LOOP }
  for pixel_y := 1 to picture.screen_max_y do begin

    for pixel_x := 1 to picture.screen_max_x do begin

{ DETERMINE VIEW RAY DIRECTION }
      x := pixel_x - picture.view_position_x;
      y := pixel_y - picture.view_position_y;
      z := pixel_z - picture.view_position_z;
      dist := sqrt(sqr(x) + sqr(y) + sqr(z));
{ CONVERT IT TO A UNIT VECTOR }
      unitx := x / dist;
      unity := y / dist;
      unitz := z / dist;

{ INITIALIZE VIEW RAY }
      initial_ray_type   := view;
      initial_ray_origin_x := pixel_x;
      initial_ray_origin_y := pixel_y;
      initial_ray_origin_z := pixel_z;
```

```
initial_ray_vector_x := unitx;
initial_ray_vector_y := unity;
initial_ray_vector_z := unitz;
initial_ray_stype    := none;
initial_intersection_flag  := false;
initial_obj_idx          := 0;
initial_subobj_idx       := 0;
initial_cpart_idx        := 0;
initial_polygon_idx      := 0;
initial_intersection_x   := 0.0;
initial_intersection_y   := 0.0;
initial_intersection_z   := 0.0;
initial_d    := 0.0;
initial_I_tr := 0.0;
initial_I_tg := 0.0;
initial_I_tb := 0.0;
initial_I_sr := 0.0;
initial_I_sg := 0.0;
initial_I_sb := 0.0;

ray_generation_number := 0;

push( initial_ray_type,
    initial_ray_origin_x,initial_ray_origin_y,initial_ray_origin_z,
    initial_ray_vector_x,initial_ray_vector_y,initial_ray_vector_z,
    initial_ray_stype,
    initial_intersection_flag,
    initial_obj_idx,
    initial_subobj_idx,
    initial_cpart_idx,
    initial_polygon_idx,
    initial_intersection_x,
    initial_intersection_y,
    initial_intersection_z,
    initial_d,
    initial_I_tr, initial_I_tg, initial_I_tb,
    initial_I_sr, initial_I_sg, initial_I_sb,
    ray_top          );


{ BEGIN RAY TRACING LOOP }
    repeat

      pop (current_ray_type.
        current_ray_origin_x,current_ray_origin_y,current_ray_origin_z,
        current_ray_vector_x,current_ray_vector_y,current_ray_vector_z,
        current_ray_stype,
        current_intersection_flag,
        current_obj_idx,
        current_subobj_idx,
        current_cpart_idx,
        current_polygon_idx,
```

```
            current_intersection_x,
            current_intersection_y,
            current_intersection_z,
            current_d,
            current_I_tr, current_I_tg, current_I_tb,
            current_I_sr, current_I_sg, current_I_sb,
            ray_top      );

        if (current_intersection_flag) then begin
{
  IF THIS FLAG IS SET THEN THIS RAY HAS ALREADY BEEN THROUGH THE RAY
 TRACING PROCESS AND HAS HIT AN OBJECT.
}
          cpart_path := picture.
                    objects^[current_obj_idx].
                    sub_objects^[current_subobj_idx].
                    common_parts;

          subobj_path := picture.
                    objects^[current_obj_idx].
                    sub_objects;

          if subobj_path^[current_subobj_idx].subobj_type = 0 then  begin
{
  IF THE SUBOBJECT TYPE IS A SPHERE THEN THE SURFACE NORMAL AT THE POINT
 OF INTERSECTION MUST BE CALCULATED SINCE IT CAN NOT BE STORED. THE SURFACE
 NORMAL IS DETERMINED FOR THE PLANE TANGENT TO THE SPHERE AT THE
 INTERSECTION POINT.
}
            surface_normal_x := current_intersection_x -
                        subobj_path^[current_subobj_idx].
                        sub_bsphere_x;

            surface_normal_y := current_intersection_y -
                        subobj_path^[current_subobj_idx].
                        sub_bsphere_y;

            surface_normal_z := current_intersection_z -
                        subobj_path^[current_subobj_idx].
                        sub_bsphere_z;

            dist := ( sqrt( sqr(surface_normal_x) +
                        sqr(surface_normal_y) +
                        sqr(surface_normal_z)));

{ THIS RAY IS THEN CONVERTED INTO A UNIT VECTOR }
            surface_normal_x := surface_normal_x / dist;
            surface_normal_y := surface_normal_y / dist;
            surface_normal_z := surface_normal_z / dist;
           end
          else begin
{
```

```
  IF THE SUBOBJECT IS A POLYGONAL OBJECT THEN THE SURFACE NORMALS FOR EACH
 OF THE POLYGONS OF WHICH IT IS COMPOSED IS RETRIEVED FROM IT'S RECORD.
}
          surface_normal_x := cpart_path^[current_cpart_idx].
                              polygons^[current_polygon_idx].
                              surface_normal_x;

          surface_normal_y := cpart_path^[current_cpart_idx].
                              polygons^[current_polygon_idx].
                              surface_normal_y;

          surface_normal_z := cpart_path^[current_cpart_idx].
                              polygons^[current_polygon_idx].
                              surface_normal_z;
        end;

{
 PROCEDURE FOR DETERMINING THE INTENSITY OF LIGHT AT EACH INTERSECTION
 POINT.
}
        I2 (current_d,
          current_I_tr, current_I_tg, current_I_tb,
          current_I_sr, current_I_sg, current_I_tb,
          picture.num_lights,
          picture.ambient_r, picture.ambient_g, picture.ambient_b,
          cpart_path^[current_cpart_idx].K_ar,
          cpart_path^[current_cpart_idx].K_ag,
          cpart_path^[current_cpart_idx].K_ab,
          cpart_path^[current_cpart_idx].K_dr,
          cpart_path^[current_cpart_idx].K_dg,
          cpart_path^[current_cpart_idx].K_db,
          cpart_path^[current_cpart_idx].K_sr,
          cpart_path^[current_cpart_idx].K_sg,
          cpart_path^[current_cpart_idx].K_sb,
          cpart_path^[current_cpart_idx].K_tr,
          cpart_path^[current_cpart_idx].K_tg,
          cpart_path^[current_cpart_idx].K_tb,
          current_ray_type,
          ray_top,
          picture.lights,
          picture.objects,
          cpart_path^[current_cpart_idx].obj_phong_exp,
          current_ray_vector_x,
          current_ray_vector_y,
          current_ray_vector_z,
          current_intersection_x,
          current_intersection_y,
          current_intersection_z,
          surface_normal_x, surface_normal_y, surface_normal_z,
          ray_generation_number,
          intensity_red,
          intensity_green,
```

```
        intensity_blue);

    end
    else begin

{ CHECK FOR POSSIBLE INTERSECTIONS OF CURRENT RAY WITH OBJECTS IN SCENE }
    check_for_intersection(current_ray_vector_x,
                           current_ray_vector_y,
                           current_ray_vector_z,
                           current_ray_origin_x,
                           current_ray_origin_y,
                           current_ray_origin_z,
                           picture.objects,
                           current_intersection_x,
                           current_intersection_y,
                           current_intersection_z,
                           current_obj_idx,
                           current_subobj_idx,
                           current_cpart_idx,
                           current_polygon_idx,
                           current_intersection_flag);


{ SET UP PATHNAMES TO USE AS SHORTHAND }
    subobj_path := picture.
                   objects^[current_obj_idx].
                   sub_objects;

    cpart_path := picture.
                  objects^[current_obj_idx].
                  sub_objects^[current_subobj_idx].
                  common_parts;

    if subobj_path^[current_subobj_idx].subobj_type = 0 then  begin
      surface_normal_x := current_intersection_x -
                          subobj_path^[current_subobj_idx].
                          sub_bsphere_x;
      surface_normal_y := current_intersection_y -
                          subobj_path^[current_subobj_idx].
                          sub_bsphere_y;
      surface_normal_z := current_intersection_z -
                          subobj_path^[current_subobj_idx].
                          sub_bsphere_z;
      dist := ( sqrt( sqr(surface_normal_x) +
                  sqr(surface_normal_y) +
                  sqr(surface_normal_z)));

      surface_normal_x := surface_normal_x / dist;
      surface_normal_y := surface_normal_y / dist;
      surface_normal_z := surface_normal_z / dist;
     end
    else begin
```

```
        surface_normal_x := cpart_path^[current_cpart_idx].
                            polygons^[current_polygon_idx].
                            surface_normal_x;

        surface_normal_y := cpart_path^[current_cpart_idx].
                            polygons^[current_polygon_idx].
                            surface_normal_y;

        surface_normal_z := cpart_path^[current_cpart_idx].
                            polygons^[current_polygon_idx].
                            surface_normal_z;
    end;

    if (current_intersection_flag) then begin
{ IF THERE HAS BEEN AN INTERSECTION THEN CONTINUE TRACING THE RAY }

        if (stack_exceeded( ray_generation_number,
                            maximum_size_of_stack)) then begin
{ IF THE STACK IS ALREADY FULL THEN CALCULATE INTENSITY AT LAST NODE }

            I6 (current_d,
                current_I_tr, current_I_tg, current_I_tb,
                current_I_sr, current_I_sg, current_I_tb,
                picture.num_lights,
                picture.ambient_r,
                picture.ambient_g,
                picture.ambient_b,
                cpart_path^[current_cpart_idx].K_ar,
                cpart_path^[current_cpart_idx].K_ag,
                cpart_path^[current_cpart_idx].K_ab,
                cpart_path^[current_cpart_idx].K_dr,
                cpart_path^[current_cpart_idx].K_dg,
                cpart_path^[current_cpart_idx].K_db,
                cpart_path^[current_cpart_idx].K_sr,
                cpart_path^[current_cpart_idx].K_sg,
                cpart_path^[current_cpart_idx].K_sb,
                cpart_path^[current_cpart_idx].K_tr,
                cpart_path^[current_cpart_idx].K_tg,
                cpart_path^[current_cpart_idx].K_tb,
                current_ray_type,
                ray_top,
                picture.lights,
                picture.objects,
                cpart_path^[current_cpart_idx].obj_phong_exp,
                current_ray_vector_x,
                current_ray_vector_y,
                current_ray_vector_z,
                current_intersection_x,
                current_intersection_y,
                current_intersection_z,
                surface_normal_x,
                surface_normal_y,
```

```
                surface_normal_z,
                intensity_red,
                intensity_green,
                intensity_blue)
            end
            else begin
{
 IF THERE WAS AN INTERSECTION AND THE STACK WAS NOT FULL THEN CALCULATE
 THE DISTANCE BETWEEN THE RAY'S ORIGIN AND POINT OF INTERSECTION AND PLACE
 THE RAY BACK ON THE STACK.
}

            current_d := (sqrt(sqr(current_intersection_x -
                            current_ray_origin_x)) +
                        (sqr(current_intersection_x -
                            current_ray_origin_x)) +
                        (sqr(current_intersection_x -
                            current_ray_origin_x)));

            push( current_ray_type,
                current_ray_origin_x,
                current_ray_origin_y,
                current_ray_origin_z,
                current_ray_vector_x,
                current_ray_vector_y,
                current_ray_vector_z,
                current_ray_stype,
                current_intersection_flag,
                current_obj_idx,
                current_subobj_idx,
                current_cpart_idx,
                current_polygon_idx,
                current_intersection_x,
                current_intersection_y,
                current_intersection_z,
                current_d,
                current_I_tr,
                current_I_tg,
                current_I_tb,
                current_I_sr,
                current_I_tg,
                current_I_tb,
                ray_top          );

{ DETERMINE IF A REFLECTED RAY WAS CREATED AND IF SO CALCULATE IT }
            calculate_reflected_ray (current_ray_vector_x,
                                current_ray_vector_y,
                                current_ray_vector_z,
                                surface_normal_x,
                                surface_normal_y,
                                surface_normal_z,
                                reflected_ray_x,
```

```
                        reflected_ray_y,
                        reflected_ray_z,
                        reflected_ray );


{ DETERMINE IF A REFRACTED RAY WAS CREATED AND IF SO CALCULATE IT }
        calculate_refracted_ray (current_ray_vector_x,
                        current_ray_vector_y,
                        current_ray_vector_z,
                        surface_normal_x,
                        surface_normal_y,
                        surface_normal_z,
                        cpart_path^[current_cpart_idx].
                         obj_refraction_index,
                        picture.global_refraction_index,
                        refracted_ray_x,
                        refracted_ray_y,
                        refracted_ray_z,
                        refracted_ray  );

        ray_generation_number := ray_generation_number + 1;

        if reflected_ray then begin

{
  IF A REFLECTED RAY WAS CREATED THEN INITIALIZE IT AND PUSH IT ON THE
 STACK.
}
          source_ray_type := current_ray_type;

          dist := ( sqrt(sqr(reflected_ray_x) +
                    sqr(reflected_ray_y) +
                    sqr(reflected_ray_z)));

{ CONVERT REFLECTED RAY TO A UNIT VECTOR }
          unitx := reflected_ray_x / dist;
          unity := reflected_ray_y / dist;
          unitz := reflected_ray_z / dist;

          initial_ray_type     := reflected;
          initial_ray_origin_x := current_intersection_x;
          initial_ray_origin_y := current_intersection_y;
          initial_ray_origin_z := current_intersection_z;
          initial_ray_vector_x := unitx;
          initial_ray_vector_y := unity;
          initial_ray_vector_z := unitz;
          initial_ray_stype    := source_ray_type;
          initial_intersection_flag  := false;
          initial_obj_idx          := 0;
          initial_subobj_idx       := 0;
          initial_cpart_idx        := 0;
          initial_polygon_idx      := 0;
```

```
                initial_intersection_x    := 0.0;
                initial_intersection_y    := 0.0;
                initial_intersection_z    := 0.0;
                initial_d       := 0.0;
                initial_I_tr    := 0.0;
                initial_I_tg    := 0.0;
                initial_I_tb    := 0.0;
                initial_I_sr    := 0.0;
                initial_I_sg    := 0.0;
                initial_I_sb    := 0.0;

                push( initial_ray_type,
                    initial_ray_origin_x,
                    initial_ray_origin_y,
                    initial_ray_origin_z,
                    initial_ray_vector_x,
                    initial_ray_vector_y,
                    initial_ray_vector_z,
                    initial_ray_stype,
                    initial_intersection_flag,
                    initial_obj_idx,
                    initial_subobj_idx,
                    initial_cpart_idx,
                    initial_polygon_idx,
                    initial_intersection_x,
                    initial_intersection_y,
                    initial_intersection_z,
                    initial_d,
                    initial_I_tr, initial_I_tg, initial_I_tb,
                    initial_I_sr, initial_I_sg, initial_I_sb,
                    ray_top          );

            end;


            if refracted_ray then begin
{
  IF A REFRACTED RAY WAS CREATED THEN INITIALIZE IT AND PUSH IT ON THE
 STACK.
}

                source_ray_type := current_ray_type;

                dist := ( sqrt(sqr(refracted_ray_x) +
                          sqr(refracted_ray_y) +
                          sqr(refracted_ray_z)));

{ CONVERT IT TO A UNIT VECTOR }
                unitx := refracted_ray_x / dist;
                unity := refracted_ray_y / dist;
                unitz := refracted_ray_z / dist;
```

```
        initial_ray_type     := refracted;
        initial_ray_origin_x := current_intersection_x;
        initial_ray_origin_y := current_intersection_y;
        initial_ray_origin_z := current_intersection_z;
        initial_ray_vector_x := unitx;
        initial_ray_vector_y := unity;
        initial_ray_vector_z := unitz;
        initial_ray_stype    := source_ray_type;
        initial_intersection_flag  := false;
        initial_obj_idx          := 0;
        initial_subobj_idx        := 0;
        initial_cpart_idx         := 0;
        initial_polygon_idx       := 0;
        initial_intersection_x    := 0.0;
        initial_intersection_y    := 0.0;
        initial_intersection_z    := 0.0;
        initial_d        := 0.0;
        initial_I_tr     := 0.0;
        initial_I_tg     := 0.0;
        initial_I_tb     := 0.0;
        initial_I_sr     := 0.0;
        initial_I_sg     := 0.0;
        initial_I_sb     := 0.0;

        push( initial_ray_type,
            initial_ray_origin_x,
            initial_ray_origin_y,
            initial_ray_origin_z,
            initial_ray_vector_x,
            initial_ray_vector_y,
            initial_ray_vector_z,
            initial_ray_stype,
            initial_intersection_flag,
            initial_obj_idx,
            initial_subobj_idx,
            initial_cpart_idx,
            initial_polygon_idx,
            initial_intersection_x,
            initial_intersection_y,
            initial_intersection_z,
            initial_d,
            initial_I_tr, initial_I_tg, initial_I_tb,
            initial_I_sr, initial_I_sg, initial_I_sb,
            ray_top          );


      end;

    end;
  end
else begin
```

```
            if (current_ray_type = view) then begin
{
 IF THERE WAS NO INTERSECTION AND THE CURRENT RAY IS THE VIEW RAY THEN SET
 THE OUTPUT INTENSITY TO THE BACKGROUND INTENSITY.
}
                intensity_red   := picture.background_color_r;
                intensity_green := picture.background_color_g;
                intensity_blue  := picture.background_color_b;

              end;
            end;

          end;

        until (stack_empty(ray_top));
{ OUTPUT THE FINAL INTENSITY }
          display_pixel( intensity_red,
                         intensity_green,
                         intensity_blue,
                         pixel_x,
                         pixel_y,
                         pixel_z  );
      end
    end;

    close (sysin);
    close (output);

end. { MAIN }
```

APPENDIX B – INPUT FILE

```
100   /x/ view position          { PICTURE }
100   /y/
1000  /z/
0.0   /red/ background light
0.0   /green/
1.0   /blue/
200   /x/ screen
200   /y/
1.0   /red/  ambient intensity
1.0   /green/
1.0   /blue/
1.0   /global refraction index/
1     /num_lights/
1.0   /red/ intensity of light source
1.0   /green/
1.0   /blue/
0.0   /x/  position of light source
20.0  /y/
0.0   /z/
0.0   /dimension1/
0.0   /dimension2/
3     /num_objects/
9999  /opcode/        { OBJECT 1 }
35.0  /radius of object's bounding sphere/
110.0 /x/ center of bounding sphere
0.0   /y/
-50.0 /z/
1     /number of subobjects/
1     /subobject type/  {SUBOBJECT 1}
35.0  /radius of subobjects bounding sphere/
110.0 /x/ center of bounding sphere
0.0   /y/
-50.0 /z/
1     /num-common-parts/  { COMMON PART 1 }
0.8   /Ka-red/      ambient coefficient
0.0   /Ka-green/
0.0   /Ka-blue/
0.8   /Kd-red/      diffuse coefficient
0.0   /Kd-green/
0.0   /Kd-blue/
0.8   /Ks-red/      specular coefficient
0.8   /Ks-green/
0.8   /Ks-blue/
0.0   /Kt-red/      transmission coefficient
0.0   /Kt-green/
0.0   /Kt-blue/
```

```
0.0   /obj-refraction-index/
200   /obj-phong-specular-exponent/
6     /num-polygons/
4     /num-vertices/  { POLYGON 1 }
90.0  /poly1 pt1/
20.0
-30.0
90.0  /poly1 pt2/
-20.0
-30.0
130.0 /poly1 pt3/
-20.0
-30.0
130.0 /poly1 pt4/
20.0
-30.0
0.0   /poly1 surface normal/
0.0
1.0
4     /num-vertices/  { POLYOGN 2 }
90.0  /poly2 pt1/
-20.0
-30.0
90.0  /poly2 pt2/
-20.0
-70.0
130.0 /poly2 pt3/
-20.0
-70.0
130.0 /poly2 pt4/
-20.0
-70.0
0.0   /poly2 surface normal/
-1.0
0.0
4     /num-vertices/  {POLYGON 3}
130.0 /poly3 pt1/
20.0
-70.0
130.0 /poly3 pt2/
-20.0
-70.0
90.0  /poly3 pt3/
-20.0
-70.0
90.0  /poly3 pt4/
20.0
-70.0
0.0   /poly3 surface normal/
0.0
-1.0
4     /num-vertices/  { POLYGON 4 }
```

```
130.0  /poly4 pt1/
20.0
-30.0
130.0  /poly4 pt2/
20.0
-70.0
90.0   /poly4 pt3/
20.0
-70.0
90.0   /poly4 pt4/
20.0
-30.0
0.0    /poly4 surface normal/
1.0
0.0
4      /num-vertices/  { POLYGON 5 }
130.0  /poly5 pt1/
20.0
-30.0
130.0  /poly5 pt2/
-20.0
-30.0
130.0  /poly5 pt3/
-20.0
-70.0
130.0  /poly5 pt4/
20.0
-70.0
1.0    /poly5 surface normal/
0.0
0.0
4      /num-vertices/  { POLYGON 6 }
90.0   /poly6 pt1/
20.0
-70.0
90.0   /poly6 pt2/
-20.0
-70.0
90.0   /poly6 pt3/
-20.0
-30.0
90.0   /poly6 pt4/
20.0
-30.0
-1.0   /poly6 surface normal/
0.0
0.0
9999   /opcode/              *** OBJECT 2 ***
175    /radius of the objects bounding sphere/
100.0  /x/ center point of the bounding sphere
-100.0 /y/
-100.0 /z/
```

```
1       /num-subobjects/
1       /subobject-type/      { SUBOBJECT 1 }
175     /radius of subobjects bounding sphere/
100.0   /x/ center of bounding sphere
-100.0  /y/
-100.0  /z/
1       /number common parts/ { COMMON PART 1 }
0.0     /red/ Ka  ambient coefficient
0.7     /green/
0.0     /blue/
0.0     /red/ Kd  diffuse coefficient
0.7     /green/
0.0     /blue
0.8     /red/ Ks  specular coefficient
0.8     /green/
0.8     /blue/
0.0     /red/ Kt  transmission coefficient
0.0     /green/
0.0     /blue/
0.0     /objects refraction index/
200     /Phong's specular exponent/
1       /number of polygons/
4       /number of vertices/    { POLYGON 1 }
0.0     /poly1 pt1/
20.0
-200.0
0.0     /poly1 pt2/
0.0
0.0
200.0   /poly1 pt3/
0.0
0.0
200.0   /poly1 pt4/
20.0
-200.0
0.0     /poly1 surface normal/
0.99
0.1
9999    /opcode/        **** OBJECT 3 ****
40      /radius of objects bounding sphere/
140     /x/ center of bounding sphere
30      /y/
-150    /z/
1       /number of subobjects/
0       /subobject type/
40      /radius of subobjects bounding sphere/
140     /x/ center of bounding sphere
30      /y/
-150    /z/
1       /number of common parts/  { COMMON PART 1 }
0.5     /red/ Ka   ambient coefficient
0.0     /green/
```

```
0.5   /blue/
0.5   /red/ Kd   diffuse coefficient
0.0   /green/
0.5   /blue/
0.8   /red/ Ks   specular coefficient
0.8   /green/
0.8   /blue/
0.0   /red/ Kt   transmission coefficient
0.0   /green/
0.0   /blue/
0.0   /refraction index for object/
200   /Phong's specular exponent/
```

LIST OF REFERENCES

1. Rogers, David F., *Procedural Elements for Computer Graphics,* McGraw-Hill, 1985.

2. Cook, Robert L., Porter, Thomas, and Carpenter, Loren, "Distributed Ray Tracing," *Computer Graphics,* v. 18, no. 3, pp. 137-145, July 1984.

3. Falby, John S., *A Data Structure for a Multi-Illumination Model Renderer,* Master's Thesis, Naval Postgraduate School, Monterey, California, December 1986.

4. Whitted, Turner, "An Illumination Model for Shaded Display," *Communications of the ACM,* v. 23, no.6, pp. 343-349, June 1980.

5. Kay, Douglas S., *Transparency, Refraction and Ray Tracing for Computer Synthesized Images,* Master's Thesis, Cornell University, Ithaca, New York, January 1979.

6. Kay, Douglas S., "Transparency for Computer Synthesized Images," *Computer Graphics,* v. 13, pp. 158-164, July 1979.

7. Falby, John S., Personal Communication, 1-30 November 1986.

8. Hecht, Eugene, *Schaum's Outline Series Theory and Problems of Optics,* McGraw-Hill, 1975.

9. Wier Maurice D., Personal Communication, 10 April - 8 May 1987.

10. Bui-Tuong, Phong, *Illumination for Computer Generated Images,* Doctoral Thesis. University of Utah, Salt Lake City, 1973.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, DC 20350-2000	2
3.	Superintendent Attn: Library (Code 0142) Naval Postgraduate School Monterey, California 93943-5002	2
4.	Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	Computer Technology Curricular Officer (Code 37) Naval Postgraduate School Monterey, California 93943	1
6.	Michael J. Zyda (Code 52Zk) Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
7.	Paul G. Smith R.D.#3 Butler, Pennsylvania 16001	1